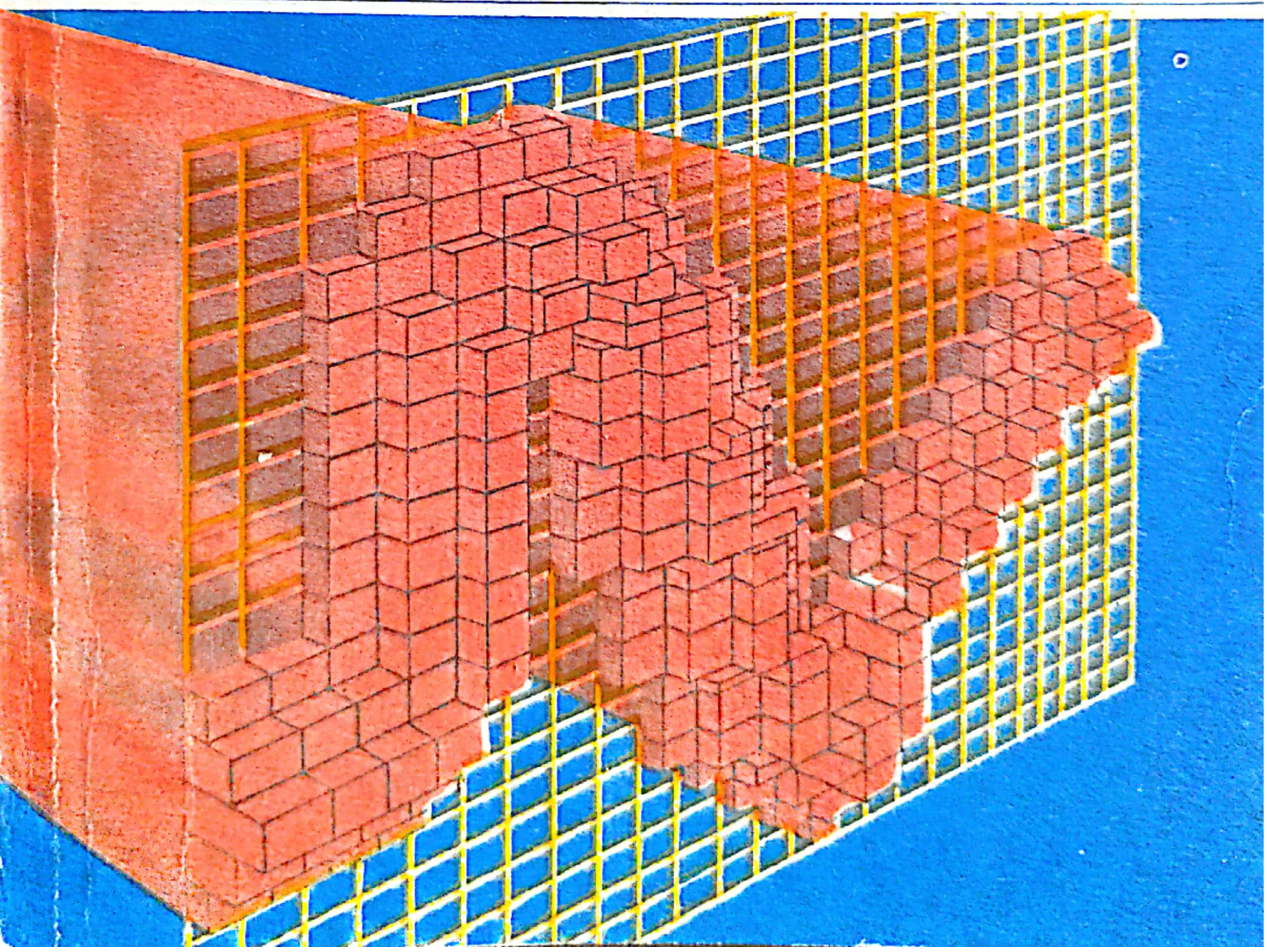
MARCEL POPESCU ION FLORICICĂ MIHAI JITARU

Metode avansate de utilizare a tehnicii de calcul



Scanned with OKEN Scanner

MARCEL POPESCU • ION FLORICICĂ • MIHAI JITARU

Metode avansate de utilizare a tehnicii de calcul

BIBLIOTECA PERSONALĂ
Nr. .....
Danilov Vasile Viorel

EDITURA MILITARĂ, BUCUREȘTI, 1989

Scanned with OKEN Scanner

Coperta: VICTOR ILIE

Scanned with OKEN Scanner

ISBN 973-32-0061-1

PREFAȚĂ

Progresele tehnologice din ultimul timp au întărit convingerea că industria de tehnică de calcul a devenit vitală pentru dezvoltarea economică a unei națiuni. Această industrie, care utilizează o cantitate mică de energie, produce valori prin transformarea unei resurse inepuizabile a omenirii, inteligența și cunoștințele umane, în această activitate nefiind nevoie în mod practic de un consum de materii prime nerecuperabile.

Se dezvoltă într-un ritm rapid industria informației, accepțiunea de „industrie" fiind strîns legată de industria calculatoarelor, de dezvoltarea lor fără precedent în istorie, nu numai prin creșterea puterii de prelucrare și miniaturizare, ci și sub aspectul accesibilității economice.

Diversificarea tipurilor de echipamente de calcul este în strînsă legătură și dependență cu diversificarea tipurilor de aplicații ale tehnicii de calcul. În acest sens, un aspect major este cel al programării calculatoarelor și implicit a limbajelor de programare. De la introducerea lor în anii '50, limbajele de programare nu au încetat să se dezvolte, să se diversifice și să se modernizeze. Fiecare limbaj reflectă mai mult sau mai puțin profunzimea la care s-a ajuns în înțelegerea activității de programare în momentul conceperii limbajului respectiv. În același timp fiecare limbaj răspunde cerințelor unei anumite categorii de utilizatori care dezvoltă un gen specific de aplicații. Astăzi însă a devenit tot mai dificil să se dea o clasificare rigidă a aplicațiilor, avînd în vedere că fiecare aplicație cu un grad sporit de dificultate implică aspecte diverse: numeric, simbolic, grafic, etc.

Evoluția spre un limbaj de programare „universal" nu poate fi evident, cheia depășirii dificultăților. Crearea unor medii de programare puternice, orientate spre categorii specifice de utilizatori, utilizarea combinată a facilităților oferite de aceste medii pentru rezolvarea de probleme complexe, pare a fi calea urmată astăzi în abordarea aplicațiilor.

Lucrarea pe care o recomandăm cititorilor este constituită din trei părți distincte: prima, dedicată prezentării limbajului C, partea a-doua — limbajului MODULA-2, partea a treia fiind consacrată limbajului PROLOG.

Deși în esență corespunde programării algoritmice clasice, totuși — prin caracteristicile sale — limbajul C impune idei noi, dintre care menționăm: portabilitatea programelor, posibilitățile de reutilizare a codului sursă, dezvoltarea de programe prin compunerea de obiecte etc. De asemenea, limbajul creează medii de programare de mare eficiență, calitate și fiabilitate. Codul obiect generat oferă posibilitatea accesului la resursele hardware ale unei mașini reale, fără a gîndi în termenii specifici limbajelor de asamblare.

Limbajul MODULA-2 este unul din primele limbaje de nivel înalt, ale cărui instrucțiuni au fost implementate hardware. Apariția lui a fost impusă de evoluția permanentă a tehnicilor de programare și de necesitatea de valorificare a creșterii spectaculoase a performanțelor calculatoarelor: memorie, viteză de execuție, periferice rapide și extrem de diversificate.

Scanned with OKEN Scanner

Elaborarea aplicațiilor complexe impune deci criterii specifice ingineriei programării: de definire, dar și de evaluare a relației cost-performanțe. Așa cum arată numeroși specialiști, complexitatea unui program crește exponențial cu dimensiunea lui, astfel că — și teoretic și practic — este mult mai ușor de a scrie și depana 25 de programe de 2 000 linii fiecare, decît a scrie și depana un singur modul de 50 000 linii de cod sursă. Ca urmare, rezultate semnificative în scăderea raportului cost/dezvoltare-întreținere se obțin prin elaborarea sistemelor software formate din grupe de module în interacțiune reciprocă.

MODULA-2, limbaj algoritmic prin excelență, promovează stilul de programare structurată și modulară, este puternic tipizat și rezolvă cu succes probleme de concurență și control în timp real al proceselor.

Limbajul PROLOG s-a dovedit a fi un instrument foarte eficace în soluționarea problemelor specifice domeniului inteligenței artificiale: demonstrarea automată de teoreme, sisteme-expert, calcul simbolic, analiza limbajului natural etc.

În principiu, limbajul PROLOG se utilizează pentru rezolvarea de probleme ce implică obiecte și relații între obiecte, avînd un fundament matematic bazat pe logica predicatelor de ordinul întîi, restrînsă la clauze Horn și principiul rezoluției al lui Robinson.

Programarea în PROLOG este diferită de cea în limbajele clasice — — FORTRAN, COBOL, ALGOL etc. —, deoarece în loc de a descrie un algoritm, adică modul de execuție pas cu pas a calculului necesar soluționării unei probleme, programatorul specifică fapte despre obiecte și relații între obiecte, după care pune întrebări referitoare la validitatea unor relații sau fapte ce rezultă din inferența relațiilor specificate. PROLOG apare deci ca un limbaj cu un pronunțat caracter descriptiv, deși are și unele caracteristici imperative. El constituie un prim pas către un nou stil de programare — programarea logică —, ce sintetizează tendințele actuale apărute în domeniul inteligenței artificiale, bazelor de cunoștințe și proceselor de învățare.

Desigur, prima întrebare firească ce se poate formula este „de ce au fost reunite în aceeași lucrare trei limbaje de programare așa de diferite, ca PROLOG, C și MODULA-2?".

Întrebarea este cu atît mai pertinentă, cu cît fiecare din aceste limbaje corespunde unui stil de programare, mai exact unei clase de metode de rezolvare a problemelor în informatică.

Un răspuns plauzibil este că această îngemănare nu este artificială întrucît numeroasele probleme ce trebuie programate astăzi și cu atît mai mult în viitor pot implica toate aspectele menționate mai sus — modularizare, concurență, algoritmizare, inteligență artificială. Într-un asemenea context folosirea combinată a acestor limbaje în aplicații informatice poate constitui o soluție practică și eficientă.

Această cale de abordare presupune competență în domenii multiple din partea proiectanților și programatorilor. Am putea vorbi chiar de o interdis-

Scanned with OKEN Scanner

ciplinaritate, dacă considerăm că prin complexitatea lor inteligența artificială, programarea concurentă și programarea algoritmică constituie deja domenii bine conturate, individualizate în programare.

Sînt argumente care pledează în favoarea ideii autorilor de a înmănunchea în paginile aceluiași volum caracteristicile fundamentale ale celor trei limbaje, și cu scopul — nemărturisit poate, dar perfect validat de realitățile acestui prezent dinamic — de a stimula fie și numai prin această modalitate evoluția de care aminteam înainte. Adăugăm și faptul că — deși implementate pe unele dintre calculatoarele electronice produse în țara noastră —, limbajele abordate în lucrare sînt mai puțin cunoscute de specialiștii informaticieni, astfel că fiecare prezentare în parte aduce un plus de cunoștințe și implicit, de competență pentru toți cei pasionați ori numai interesați de evoluția informaticii.

Dr. ing. VASILE BALTAC

PARTEA I

PROGRAMARE ÎN LIMBAJ C

CAPITOLUL 1

INTRODUCERE

Dezvoltarea inițială a limbajului de programare C — proiectat de Dennis Ritchie în jurul anului 1972 în laboratoarele Bell ale companiei AT&T (SUA) a fost strîns legată de sistemul de operare UNIX, deoarece atît sistemul, cît și cea mai mare parte a programelor care rulează sub acesta erau scrise în C.

Sistemul de operare UNIX a fost conceput ca un sistem portabil, care punea la dispoziția utilizatorilor un set larg de instrumente software pentru dezvoltarea de programe în conformitate cu principiile ingineriei programării. Între aceste instrumente, limbajul C a fost gîndit în ideea de a ocupa locul cel mai important.

Deși C a fost considerat ca un „limbaj de programare de sistem", deoarece este adecvat pentru scriere de sisteme de operare, compilatoare, editoare de texte și alte programe din această categorie, el poate fi considerat și ca un limbaj de programare de uz general. Practic, C a fost și este folosit pentru a dezvolta programe în cele mai diverse domenii. Calitățile sale — fiabilitatea, regularitatea, simplitatea, ușurința în învățare și utilizare — au determinat ca acest limbaj să fie preferat de mulți programatori. În prezent este implementat ca sistem independent de dezvoltare de programe pe aproape toate tipurile de calculatoare: microcalculatoare de 8/16 biți (subsisteme de operare de tip CP/M sau MS-DOS), minicalculatoare de 16/32 biți (subsisteme de operare compatibile RSX 11M sau VMS) etc.

Scanned with OKEN Scanner

Multe din conceptele importante ale limbajului C derivă din limbaje mai vechi — Algol 60 (proiectat de un comitet internațional în 1960), CPL (*C*ombined *P*rogramming *L*anguage — proiectat în Anglia în 1963), BCPL (*B*asic *C*ombined *P*rogramming *L*anguage — proiectat de Martin Richards la Cambridge în 1967) și, în fine, B (proiectat de Ken Thompson la Laboratoarele Bell — SUA în anul 1970 pentru primul sistem UNIX pe un calculator PDP-07).

Caracteristica esențială a limbajului C este legată de faptul că oferă mijloace pentru a construi programe complexe din elemente simple. Este posibilă astfel creșterea importantă a productivității muncii de programare prin folosirea de instrumente software.

Spre deosebire de limbajele BCPL și B, C pune la dispoziție o varietate largă de tipuri de date: tipuri de bază și tipuri derivate. Tipurile de bază cuprind caracterele, întregii de diferite lungimi cu sau fără semn, numerele reale de diferite precizii și șirurile de caractere. La rîndul lor, tipurile de date derivate pot fi create într-un spectru larg, folosind pointeri, tablouri, structuri și reuniuni.

Există o diversitate largă de operatori — aritmetici, logici, relaționali, pentru manipulare biți, de asignare, de adresă etc. Expresiile se pot construi din operanzi și diversele tipuri de operatori; orice expresie, inclusiv o asignare sau un apel de funcție, poate fi termen într-o altă expresie sau instrucțiune.

Instrucțiunile pentru controlul fluxului execuției în orice limbaj specifică ordinea în care se realizează operațiile de calcul. Limbajul C cuprinde construcțiile de bază necesare scrierii de programe bine structurate în sensul teoriei „programării structurate". Astfel, există mecanisme pentru a defini blocuri (sau pentru a grupa instrucțiunile), mecanisme de selecție binară (tip *if-then-else*) sau selecție multiplă (tip *switch*), mecanisme pentru programare cicluri cu evaluarea testului de terminare înainte (tip *while, for*) sau după *(do-while)* execuția corpului acestora, mecanisme pentru controlul execuției ciclurilor *(continue, break)*.

Conceptul de *funcție*, folosit în C, ca unitate independentă de program, chiar compilabilă separat, permite structurarea programelor complexe în *module*. În acest fel proiectarea programelor C poate avea la bază principiile unor metode clasice de proiectare ca: programarea modulară, proiectarea structurată (în sensul lui Jourdon și Constantine), proiectarea compusă (Myers G.), analiza și proiectarea structurată SADT (Rosi) etc. Conceptele de *consistență* (ce caracterizează structura internă a unui modul) și *cuplaj* (ce caracterizează interfața dintre două module) își găsesc în C posibilități reale de implementare. Astfel, funcția C permite dezvoltarea de module cu *consistență funcțională* (tipul cel mai bun pe scara consistențelor), module care realizează funcții bine defi-

Scanned with OKEN Scanner

nite. În ceea ce privește cuplajul precizăm că principala formă pentru schimb de date între funcțiile C este oferită de *comunicarea prin date*. Deci, *cuplajul prin date* (tipul cel mai bun pe scara cuplajelor) constituie forma normală de comunicare de date între funcții C.

În plus, față de cele prezentate mai sus, funcțiile C permit *recursivitatea* și pot returna valori corespunzătoare tipurilor de bază (întreg, real, șir de caractere), dar și tipurilor derivate (pointeri, structuri, reuniuni).

Printr-un pas distinct de preprocesare pot fi incluse în programele curente alte fișiere cu cod sursă, se pot face substituții în codul sursă și compilări condiționate. În acest fel codul sursă al oricărui program C poate fi parametrizat.

În același timp, C este considerat ca un „limbaj de nivel relativ scăzut", în comparație cu limbaje de nivel înalt cum sînt PL/I, PASCAL, FORTRAN, COBOL sau chiar cu limbaje de nivel înalt pentru programare de sistem cum sînt, de exemplu, Ada, CLU, Mesa, Modula-2, Edison. Acest lucru este adevărat deoarece C pune la dispoziție mecanisme pentru a utiliza direct unele resurse *hardware*, cum ar fi registrele, adresele, numerele și caracterele în reprezentare internă etc. Aceste mecanisme se încadrează în filozofia generală a limbajului, fără a constitui elemente izolate. Spre exemplu, o variabilă care este frecvent folosită poate fi alocată într-un registru al mașinii reale și nu în memorie. În acest fel, în mod natural se poate controla eficiența programului.

În sfîrșit, C nu conține instrucțiuni speciale pentru operații de intrare/ieșire și nici metode specifice de acces la fișiere. Toate aceste operații se pot realiza prin apeluri explicite de funcții. Toate implementările conțin o bibliotecă cu o colecție largă de astfel de funcții, pe care programele le pot selecta în funcție de obiectivele propuse.

În momentul de față pentru limbajul C există un standard oficial elaborat de Institutul național de standarde din SUA (standard ANSI). Acest standard este bazat pe manualul de referință original publicat în lucrarea autorilor B.W. Kernighan și D. M. Ritchie — *The C Programming Language* (1978, 1988). În lucrarea de față se vor urmări liniile generale ale acestui standard, dat fiind faptul că în anii următori subsistemele de dezvoltare a programelor în limbajul C existente în țara noastră vor corespunde cu aceste precizări.

Programele prezentate ca exemple au fost verificate subsistemul de dezvoltare de programe TURBO C V 1.5, implementat pe microcalculatoare de 16 biți (sisteme de operare tip MS-DOS) sau pe implementarea C DECUS existentă pe minicalculatoare de 16 biți tip CORAL/ Independent.

BIBLIOTECA PERSONALA

Scanned with OKEN Scanner

CAPITOLUL 2

INTRODUCERE ÎN PROGRAMAREA ÎN C

2.1. Structura sintactică a unui program C

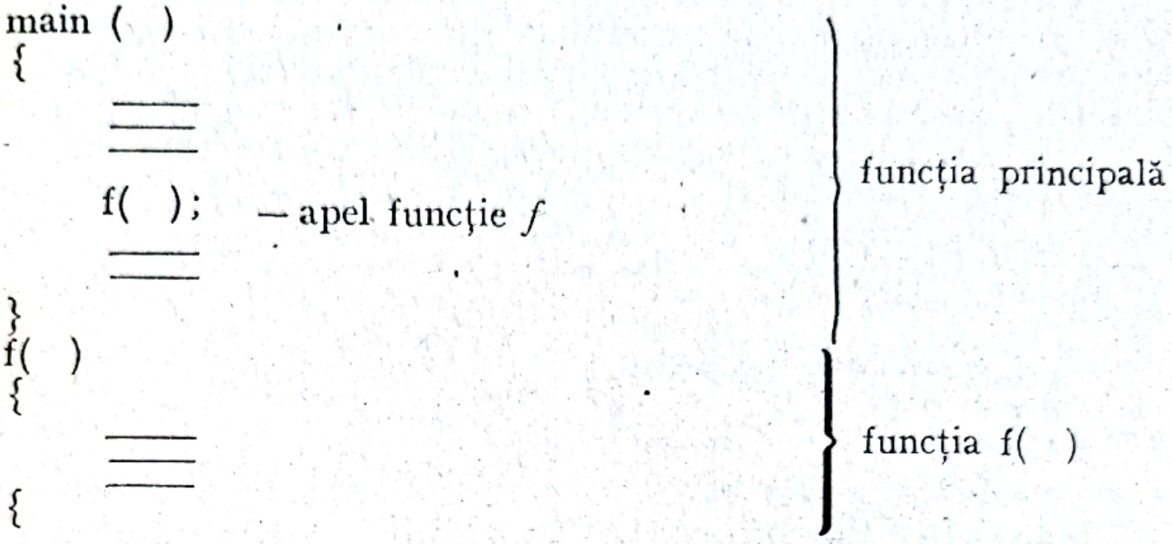
Conceptul de bază folosit în structurarea programelor scrise în limbajul C este *funcția*. Astfel, un program C este compus din cel puțin o funcție, și anume funcția *main*() sau funcția principală. La rîndul ei funcția *main*() poate apela alte funcții definite de utilizator sau existente în bibliotecile ce însoțesc orice mediu de dezvoltare de programe în C. Structura generală a unei funcții C este de forma:

```
nume-funcție(arg1, arg2, ..., argn)
_____ }
_____ }  instrucțiuni declarare tip argumente
_____ }
{
_____ }
_____ }
_____ }  corp funcție = secvență de instrucțiuni sau apeluri
_____ }                 de funcții
}
```

unde *nume-funcție* reprezintă numele sub care funcția este cunoscută în program; *arg1, arg2, ..., argn* — parametrii cu care funcția C este apelată și al căror tip poate fi declarat direct în această listă (vezi TURBO C) sau prin instrucțiuni separate imediat după lista lor; corpul funcției este definit ca o secvență de instrucțiuni și/sau apeluri de funcții și delimitat de restul funcției prin paranteze acolade. Funcțiile C sînt în general unități independente compilabile separat. Instrucțiunile, la rîndul lor, pot defini tipul unor date folosite în program sau operații ce trebuie executate prin program. Din punct de vedere sintactic orice instrucțiune trebuie terminată cu caracterul „;", iar grupuri de instrucțiuni pot fi delimitate prin caracterele „{" și „}" pentru a forma unități sintactice noi de tip *bloc*. Funcțiile apelate pot primi valori pentru argumente și transmite înapoi spre funcția apelată o valoare rezultată din efectuarea operațiilor asupra valorilor primite.

Scanned with OKEN Scanner

Cu aceste precizări generale, dacă avem un program compus din două funcții — și anume, funcția principală și o funcție apelată, $f()$ —, atunci structura acestuia va fi de forma:



Funcția *main*() este în general folosită fără parametri. Cazurile în care ea se utilizează cu parametri vor fi prezentate ulterior. La rîndul lor funcțiile apelate pot fi scrise în limbaj C sau realizate și în alte limbaje de programare: limbaje de asamblare, FORTRAN, PASCAL etc.

Pentru scrierea programelor C se folosesc de obicei litere mici; literele mari sînt folosite mai rar și în general pentru a denumi diverse constante ce apar în programe.

2.2. Exemple de programe C

Ne propunem să scriem un program C care să afișeze pe ecranul terminalului de unde se lucrează următoarea informație:

ION-ALEXANDRU, Str.-Al.-Russo, nr. 5

Pentru aceasta vom folosi din biblioteca mediului de dezvoltare de programe în C funcția *printf* (), funcție care primind ca argument un șir de caractere îl afișează pe terminalul curent. Cu această precizare programul C cerut poate fi următorul (P1.C):

```
/*
 *      Program P1.C
 */
main()
{
        printf("ION ALEXANDRU,Str. Al. Russo,nr. 5 \n");
}
```


Scanned with OKEN Scanner

Modul exact în care acest program C este transformat în program executabil pe calculator depinde de mediul C în care se lucrează. În general, cu un editor de texte se construiește un fișier pe disc cu extensia „C", care apoi este compilat și transformat în modul obiect, linkeditat și transformat în program executabil (în această fază se adaugă din bibliotecă și modulul obiect pentru funcția *printf*()), după care este lansat în execuție printr-o comandă specifică. De exemplu, pe minicalculatoare (subsisteme de operare de tip RSX11M) secvența de comenzi poate fi următoarea:

```
>EDI P1.C (CR)
main ( ) (CR)
{(CR)
   printf ("ION ALEXANDRU, Str. Al. Russo, nr. 5\n") ;(CR)
} (CR)
(CTRL/Z)
>XCC P1-A (CR)
>TKB P1/CP=P1, LB: [1, 1]C/LB(CR)
>RUN P1 (CR)
```

unde cu (CR) s-a notat terminatorul standard de linie, iar cu P1.C fișierul sursă cu programul C.

Se remarcă faptul că șirul de caractere ce trebuie afișat prin *printf*() este delimitat folosind caracterul ". În șir pot fi utilizate caractere afișabile și caractere neafișabile. Un caracter neafișabil este introdus printr-o literă precedată de simbolul „\". Așa este cazul lui „\n" care constituie o notație în C pentru caracterul NL *(newline)*. De pildă, dacă am dori ca informația de afișat să fie scrisă sub forma:

ION ALEXANDRU
Str. Al. Russo, nr. 5

atunci programul trebuie rescris sub forma (P2.C):

```
/*
 *      Program P2.C
 */
main()
{
        printf("ION ALEXANDRU\nStr. Al. Russo,nr. 5 \n");
}
```

Să modificăm acum programul anterior în sensul că informația de afișat să fie mai întîi introdusă la cererea programului prin intermediul tastaturii terminalului. Pentru a citi informația vom folosi funcția

Scanned with OKEN Scanner

scanf() din biblioteca C. Noua formă a programului poate fi următoarea (P3.C):

```
/*
 *      Program P3.C
 */
main()
{
        char nume[31],adresa[41];

        printf("\nNume: ");
        scanf("%s",nume);
        printf("\nAdresa: ");
        scanf("%s",adresa);
        printf("%s \n%s \n",nume,adresa);
}
```

În C orice dată utilizată trebuie să fie declarată. Aici s-au folosit variabilele *nume* și *adresă* care prin linia

char nume[31], adresa[41];

au fost declarate ca date de tip șir de caractere de maximum 30 și, respectiv, 40 caractere. Șirul „%s" care apare în apelul funcției *scanf*() precizează că se vor citi în variabilele *nume* și, respectiv, *adresa* valori de tip șir de caractere. Un astfel de șir de forma „%s" poartă numele de *descriptor* sau *specificator de conversie*. În funcție de descriptor citirea unor date poate fi însoțită și de o conversie de format adecvată. În *printf*() descriptorul „%s" are rolul de a preciza cum trebuie convertite valorile datelor de afișat (în cazul de față, valorile variabilelor *nume* și *adresă*).

Funcția *scanf*() citește un șir de caractere din bufferul de intrare pînă cînd întîlnește un spațiu, un caracter (TAB) sau ajunge la sfîrșitul acestuia. Astfel, dacă se tastează „ION ALEXANDRU", atunci se va memora în variabila *nume* doar valoarea „ION". Pentru a obține șirul în întregime este recomandat să se transmită numele sub forma: „ION-ALEXANDRU".

2.3. Reorientarea intrărilor și ieșirilor standard într-un program C

Orice program C lucrează cu trei dispozitive periferice standard: un dispozitiv standard de intrare *(stdin)*, unul de ieșire *(stdout)* și, respectiv, unul pentru erori *(stderr)*; implicit aceste dispozitive standard sînt conectate la terminalul de la care se lansează programul în execuție. În realitate funcția *printf*() afișează informația pe dispozitivul standard de ieșire *(stdout)*, iar funcția *scanf*() citește date de pe dispozitivul standard de intrare. La lansarea în execuție a unui program C dispozitivele *stdin* și/sau *stdout* pot fi reasignate cu alte periferice din configurație.

Scanned with OKEN Scanner

Această reasignare se poate realiza prin informații introduse în linia de comandă prin care se activează programul. Astfel, dacă în programul anterior (P2.C) se dorește ca scrierea numelui și adresei să se facă pe imprimantă, programul executabil pe un minicalculator fiind P2.TSK, atunci el va fi activat astfel:

```
≥RUN P2 (CR)
TTn>>LP: (CR)
```

Aici s-au subliniat informațiile afișate de calculator. Informația „>LP:" din linia de comandă determină reasignarea perifericului standard *stdout* cu imprimanta (LP:). Dacă dorim ca informația să fie scrisă pe disc în fișierul P2.DAT, atunci se poate proceda astfel:

```
≥RUN P2 (CR)
TTn>>P2.DAT (CR)
```

În situația că fișierul P2.DAT există deja și se intenționează ca informația nouă să fie scrisă la sfîrșitul acestuia, atunci se înlocuiește în linia de comandă șirul „>P2.DAT" cu „≫P2.DAT". Atunci cînd cele două informații ce trebuie citite prin *scanf*() în programul P3.C se află într-un fișier P3IN.DAT ele pot fi preluate activînd programul sub forma:

```
≥RUN P3(CR)
TTn> <P3IN.DAT >P3.DAT (CR)
```

În acest caz, programul P3.C trebuie rescris sub forma:

```
main()
{
    char nume [31], adresa [41];
    scanf("%s", nume);
    scanf ("%s", adresă);
    printf("%s\n% s\n", nume, adresă);
}
```

Din interiorul unui program C, citirea și afișarea de informații folosind alte dispozitive decît *stdin* și *stdout* se pot realiza folosindu-se conceptul de fișier. De exemplu, dacă în programul anterior dorim ca informațiile introduse de la terminal să fie scrise pe imprimantă, atunci programul trebuie rescris sub forma (P4.C):

De această dată s-a folosit funcția standard *fprintf*() pentru scrierea cu format a unei informații într-un fișier identificat prin variabila *fp*. Variabila *fp* conține un pointer spre tabela de descriere a fișierului în care se face operația. Acest pointer este returnat după execuția funcției *fopen*(), care deschide fișierul specificat (în cazul de față imprimanta) pentru scriere și este atribuit variabilei *fp*. Declarația:

```
FILE *fp, *fopen();
```


Scanned with OKEN Scanner

```
/*
 *      Program P4.C
 */
#include <stdio.h>

main()
{
        FILE *fp,*fopen();
        char nume[31],adresa[41];

        fp=fopen("PRN","w");                    /* deschide fisierul */
        printf("\nNume: ");
        scanf("%s",nume);                       /* citeste nume */
        printf("\nAdresa: ");
        scanf("%s",adresa);                     /* citeste adresa */
        fprintf(fp,"%s\n%s\n",nume,adresa);
        fclose(fp);                             /* inchide fisier */
```

precizează faptul că atît variabila *fp*, cît și rezultatul returnat prin execuția funcției *fopen*() sînt de tipul FILE (pointer spre o tabelă de descriere de fișier). Tipul FILE este definit în fișierul prefix *stdio.h* ce se găsește în orice mediu C. Acest fișier prefix — declarat prin directiva # *include* — este inclus în program la compilare.

Dacă datele trebuie scrise pe disc în fișierul P4.DAT, atunci în programul P4.C se modifică doar linia 6 sub forma:

fp = *fopen* (''P4.DAT'', ''w'');

Dacă un fișier trebuie citit, atunci în funcția *fopen*() în loc de modul ''w'' se folosește modul ''r''.

2.4. Variabile, expresii și atribuiri

Să realizăm acum un program care citește trei numere întregi *a*, *b*, *c*, evaluează expresia *suma* = *a* + *b* + *c* și afișează rezultatul obținut. Introducînd datele prin perifericul *stdin* și afișîndu-le pe *stdout*, codul sursă al programului cerut poate fi (P5.C):

```
/*
 *      Program P5.C
 */
main()
{
        int a,b,c,suma;

        printf("\nIntroduceti 3 numere: ");
        scanf("%d %d %d",&a,&b,&c);     /* citeste numerele */
        suma=a+b+c;
        printf("Suma este: %d \n",suma);
```


Scanned with OKEN Scanner

Prin instrucțiunea: int a, b, c, suma;

variabilele *a, b, c, suma* sînt declarate de tip întreg (*int*). Pentru citirea și afișarea valorilor lor se remarcă folosirea în *scanf*() și *printf*() a descriptorului de format %*d*, care arată că numerele citite sau scrise sînt secvențe de cifre zecimale, eventual precedate de semnul „—" (minus) în cazul numerelor negative. Notațiile &*a*, &*b*, &*c* în *scanf*() semnifică faptul că funcția cere să i se transmită adresele în memorie ale zonelor alocate variabilelor *a, b, c*. În momentul execuției programului cele trei numere se tastează, unul după altul, separate prin spații și/sau caractere (TAB).

În instrucțiunea:

$$suma = a + b + c;$$

„+" este operatorul aritmetic de adunare, iar „=" reprezintă operatorul de atribuire. Valoarea expresiei $a + b + c$ este atribuită variabilei suma, iar pe ansamblu secvența $suma = a + b + c$ este o *expresie de atribuire* cu două valori: *adevărat* — dacă valoarea atribuită este diferită de zero, și *fals* — dacă valoarea atribuită este egală cu zero.

O variabilă de tip *int* semnifică o valoare întreagă reprezentată pe 16 biți, deci practic orice număr întreg din domeniul [—32 768, +32 767]. Dacă valoarea este în afara limitelor acestui interval se poate folosi tipul *long* ce reprezintă valori întregi pe 32 biți, deci din intervalul $[-2^{31}, 2^{31}-1]$. De obicei, pentru valori întregi pe 8 biți se folosește tipul *char*, iar pentru numere reale, tipurile *float* (virgulă mobilă pe 32 biți) și *double* (virgulă mobilă pe 64 biți).

În continuare se dă un exemplu de program (P6.C) care afișează volumul (*v*) și aria totală (*s*) ale unui con de rază *r* și înălțime *h*. Valorile corespunzătoare lui *r* și *h* se citesc de pe perifericul *stdin*. De această dată se operează cu numere reale, pentru declararea lor folosindu-se tipul *float*.

```
/*
 *      . Program P6.C
 */
#include <math.h>

main()
{
        float r,h,g,v,s;

        printf("Introduceti r si h : ");
        scanf("%f %f",&r,&h);
        g=sqrt((double)(r*r+h*h));          /* calcul generatoare */
        v=(1./3.)*3.141593*r*r*h;           /* calcul volum */
        s=3.141593*r*(r+g);                 /* calcul arie totala */
        printf("r= %7.3f, h= %7.3f\n",r,h);
        printf("v= %7.3f, s= %7.3f\n",v,s);
}
```


Scanned with OKEN Scanner

Pentru citirea și scrierea valorilor reale se remarcă folosirea descriptorului „%*f*". În forma „%7.3*f*" se precizează că valorile numerelor *v* și *s* vor fi scrise cu cel puțin 7 cifre (inclusiv marca zecimală) din care 3 sînt zecimale.

S-a folosit de asemenea operatorul binar „*", pentru înmulțirea a două numere, și funcția standard *sqrt*() din biblioteca C, în scopul extragerii rădăcinii pătrate. Argumentul corespunzător funcției *sqrt*() trebuie să fie de tip *double*, din care cauză prin expresia

(double) (r * r + h * R)

s-a indicat explicit conversia valorii expresiei $r * r + h * h$ din *float* în *double*. O expresie asemănătoare cu cea de mai sus poartă numele de „*cast*" și se folosește în general pentru a forța conversia explicită a unor valori dintr-un format de reprezentare în altul (aici, din *float* în *double*). Rezultatul funcției *sqrt*() este tot de tip *double*, însă el este convertit înapoi în *float* în momentul în care variabilei *g*, de tip *float*, i se atribuie valoarea rezultată din extragerea rădăcinii pătrate.

Fișierul prefix *math.h* inclus în program prin directiva #include <math.h> declară funcțiile matematice standard ce pot fi utilizate în programe C.

În cazul de față s-a folosit doar funcția *sqcrt*().

Programul P6.C prezintă o anumită particularitate, în sensul că folosește variabile de tip *float* și *double*. Construirea programelor executabile plecîndu-se de la astfel de programe implică cîteva precizări suplimentare. De obicei, funcțiile *scanf*() și *printf*() din biblioteca C nu „știu" să interpreteze corect descriptori de tipul „%*f*" (sînt furnizate într-o formă mai simplă). Pentru a realiza corect conversiile din format *șir de caractere* în *float* și invers există două module obiect separate, numite ATOD.OBJ și DTOA.OBJ. La editarea de legături aceste module trebuie incluse explicit în programe. Astfel, pe minicalculatoare, un asemenea program executabil se obține folosindu-se o secvență de comenzi de forma:

```
>XCC  —A P6 (CR)
>TKB P6/CP/FP = P6, LB: [1, 1]ATOD, DTOA, C/LB
```

2.5. Constante simbolice

Pentru program numărul 3.141593 este o constantă. În scopul utilizării de *nume simbolice* în programe în loc de valori numerice concrete, care sînt mai „criptice", se poate folosi conceptul de *constantă simbolică*. O astfel de constantă se definește prin directiva # *define* sub forma:

define nume-simbolic constantă

Scanned with OKEN Scanner

În cazul concret al programului P6.C se poate defini constanta simbolică PI sub forma:

define PI 3.141593

Cu această inovație programul P6.C se rescrie într-o formă mai clară după cum urmează (P7.C):

```
/*
 *      Program P7.C
 */
#include <math.h>
#define  PI 3.141593

main()
{
        float r,h,g,v,s;

        printf("Introduceti r si h : ");
        scanf("%f %f",&r,&h);
        g=sqrt((double)(r*r+h*h));          /* calcul generatoare */
        v=(1./3.)*PI*r*r*h;                 /* calcul volum */
        s=PI*r*(r+g);                       /* calcul arie totala */
        printf("r= %7.3f, h= %7.3f\n",r,h);
        printf("v= %7.3f, s= %7.3f\n",v,s);
}
```

Cantitățile de genul lui PI sînt constante exprimate simbolic și nu variabile. Din această cauză, ele nu trebuie neapărat declarate. Pentru a le deosebi de variabile, ele se notează de obicei cu litere mari. Aceste constante nu există decît în textul sursă inițial al programelor. Înainte de compilarea programului constantele simbolice sînt înlocuite peste tot cu valorile asociate.

CAPITOLUL 3

ELEMENTE DE BAZĂ ALE PROGRAMĂRII ÎN C

În general, programele sînt construite pentru a rezolva probleme prin prelucrarea unor date. Procesul de soluționare, descris algoritmic într-un limbaj cum este C, cuprinde operații ca: citirea datelor de prelucrat; verificarea corectitudinii lor; memorarea lor în spațiul programului; manipularea acestora prin operații disponibile în limbaj; scrierea rezultatelor pe un suport extern pentru păstrare sau utilizare imediată.

În continuare, vom prezenta mai exact o serie de elemente din limbajul C folosite în vederea programării operațiilor de intrare/ieșire, definirii tipurilor de date, efectuării de operații de calcul, execuției condițio-

Scanned with OKEN Scanner

nate repetate de operații, definirii de funcții. Pentru descrierea exactă a unor structuri sintactice se va folosi o notație tip BNF (*Backus-Naur Form*).

3.1. Operații de ieșire

Prin astfel de operații un program scrie informații pe ecran (text și/sau grafică), pe un disc flexibil sau de alt tip, pe imprimantă sau pe un canal de comunicație (de exemplu, spre un alt calculator). În acest paragraf vor fi prezentate cîteva funcții standard din C pentru a efectua operații de ieșire. De menționat faptul că în capitolul anterior s-au folosit funcțiile *printf*() și *fprintf*() pentru scriere cu format.

Funcția printf() se folosește pentru a scrie cu format informații pe perifericul standard de ieșire *(stdout)* al programului, asignat de obicei cu ecranul terminalului curent. Funcția *printf*() are o formă simplă și flexibilă:

printf(<*șir-format*>, <*expr1*>, <*expr2*>, ...);

Primul parametru al funcției este un șir de caractere delimitat cu apostrofuri duble, pe care *printf*() încearcă să-l scrie pe *stdout*. Acest șir conține două tipuri de obiecte: caractere afișabile sau neafișabile (introduse prin *secvențe-escape*), care sînt copiate ca atare pe *stdout* și specificatori de conversie, sau descriptori de format (introduși prin caracterul „%"), folosiți pentru a converti în format afișabil valorile argumentelor <*expr1*>, <*expr2*>, ...

De exemplu, în instrucțiunea:

printf("Suma este %d\n", *suma*);

șirul „Suma este" are în compunere caractere afișabile, „\n" reprezintă caracterul neafișabil *newline* introdus printr-o secvență-escape, iar „%d" — un specificator de conversie pentru valoarea variabilei întregi *suma*.

Toți specificatorii de format încep cu caracterul „%" și se termină cu o literă care indică tipul datei și modul în care ea trebuie convertită.

Corespunzător fiecărui specificator de format este necesar să existe în apelul funcției *printf*() un element de afișat — <*expr1*>, <*expr2*>,... Elementele de afișat pot fi constante, variabile, expresii, apeluri de funcții care returnează valori sau combinații. În general, tipul elementelor de afișat trebuie să corespundă cu tipul indicat prin specificatorul de format. Expresiile *„cast"* se pot folosi pentru compatibilizarea tipului unui element de afișat cu specificatorul de format. Cîteva specificatoare de format mai des folosite sînt următoarele:

- %*d* — element de tip *int* convertit în număr zecimal cu semn;

Scanned with OKEN Scanner

- %*s* — element de tip *char* (vector de caractere terminat cu caracterul ASCII NULL — „\0'), tipărit;
- %*f* — element de tip *float* sau *double* afișat sub forma [—]mmm.nnn (6 cifre din care 3 în partea zecimală););
- %*c* — element de tip *char* (un singur caracter doar este afișat
- %*e* — element de tip *float* sau *double* afișat sub forma [—]m.nnnnnne±xx.

Între % și literă poate fi inclus și un număr ce indică dimensiunea minimă a cîmpului în care se face afișarea. De exemplu, %4*d* indică un cîmp de minimum 4 cifre, iar %7.4*f* — un cîmp de minimum 7 cifre, din care 4 după marca zecimală.

Caracterele neafișabile sînt introduse prin secvențe-escape. Citeva secvențe-escape uzuale sînt:

- \n — caracterul *newline* (LF);
- \t — caracterul HT;
- \b — caracterul *backspace* (BS);
- \r — caracterul CR;
- \„ — caracterul „apostrof dublu";
- \\ — caracterul „*back slash*";
- \0 — caracterul NULL;
- \nnn — caracterul cu codul ASCII nnn(1÷3 cifre octale);
- \xhh — caracterul cu codul ASCII hh (1÷2 cifre hexazecimale).

De exemplu, dacă se dorește afișarea șirului de caractere „Adresă:" pe linia 10 și din coloana 20 a ecranului unui terminal DAF 2020 (compatibil VT 100, mod ANSI), atunci se poate folosi următorul apel al funcției *printf*():

printf("\033[%d;%dHAdresă", 10, 20);

iar dacă înainte de afișare ecranul trebuie șters, atunci apelul

printf("\033[2J");

realizează această operație.

Funcția fprintf() este folosită pentru a scrie cu format informații într-un fișier precizat prin primul argument din apelul acesteia:

fprintf(⟨*pointer-FILE*⟩, ⟨*șir-format*⟩, ⟨*expr-1*⟩, ⟨*expr-2*⟩,...);

În programul P4.C este arătat modul de definire și inițializare a variabilei ⟨*pointer-FILE*⟩, ce definește fișierul în care se face scrierea.

Observație. Atît funcția *fprintf*(), cît și *printf*() returnează după apel numărul de caractere scris efectiv sau o valoare negativă dacă în scriere a apărut o eroare. Astfel, după un apel de forma:

l = printf("Adresă:");

variabila întreagă *l* va avea valoarea 7 sau o valoare negativă dacă scrierea nu s-a făcut corect ■

Scanned with OKEN Scanner

Funcția *puts*() permite scrierea unui șir de caractere terminat cu caracterul NULL ('\0') și a unui caracter *newline* pe fișierul *stdout*. Formatul este:

puts (⟨sir-caractere⟩) ;

După apel funcția transmite o valoare pozitivă dacă scrierea s-a făcut corect sau EOF cînd la scriere apare o eroare. Constanta simbolică EOF (de tip *int*) este definită în fișierul prefix ⟨*stdio.h*⟩, astfel că pentru a o putea referi în program este necesară includerea acestui fișier în program la compilare.

De exemplu, cu funcția *puts*() programul P2.C s-ar rescrie sub forma (P8.C):

```
/*
 *      Program P8.C
 */
main()
{
        puts("ION ALEXANDRU");
        puts("Str. Al. Russo;nr. 5");
}
```

Observație. Atunci cînd un șir de caractere de tipul „ION ALEXANDRU" este generat, la sfîrșit i se adaugă automat un caracter NULL. Ca urmare, acest șir va ocupa în memorie 14 locații de cîte 1 bait și nu 13 cît rezultă după numărul caracterelor sale ■

Funcția putchar() scrie un singur caracter, dat ca argument, pe fișierul *stdout*. Formatul ei este:

putchar (⟨*caracter*⟩) ;

după apel această funcție returnează caracterul scris sau EOF dacă scrierea nu a fost corectă. De exemplu, prin apelul

lf = putchar ('\n') ;

pe *stdout* se scrie caracterul *newline* și totodată — ca efect secundar — acest caracter este memorat și în variabila *lf* (de tip *char* sau *int*). De asemenea, în cazul unei scrieri fără erori, apelul:

putchar (putchar('\n')) ;

transmite două caractere *newline* spre *stdout*.

3.2. Tipuri de date

Atunci cînd se scriu programe C se folosesc date care se încadrează în unul din următoarele tipuri de bază (sau tipuri fundamentale): *char*, *int*, *float*, *double*, *enum*, *void*. Tipurile *char* și *int* pot fi prefixate cu *signed* și *unsigned*, tipul *int* poate fi prefixat cu *short* sau *long*, iar *double* cu *long*. Se obține astfel o varietate largă de tipuri de bază prin care datele descrise se pot pune în corespondență cu reprezentările posibile ale datelor

Scanned with OKEN Scanner

la nivelul limbajului-mașină. O parte dintre aceste tipuri au fost introduse o dată cu standardul ANSI pentru limbajul C (1988), astfel că în implementări mai vechi pot să nu existe toate variantele menționate.

În fiecare mediu de programare C, domeniul de valori posibile pentru fiecare tip este precizat în fișierele prefix ⟨*limits.h*⟩ și ⟨*float.h*⟩. În continuare, sînt prezentate cîteva implementări uzuale:

Denumire tip	*Memorie alocată* (biți)	*Domeniu de valori*
char	8	[0,127] — cod ASCII
unsigned char	8	[0,255]
signed char	8	[−128, 127]
int	16	[−32 768, 32 767]
short int	16	[−32 768, 32 767]
long int	32	[−2 147 483 648, 2 147 483 647]
unsigned int	16	[0,65535]
unsigned long	32	[0,429 496 7295]
float	32	$(-10^{37}, -10^{-37}) \cup \{0\} \cup (10^{-37}, 10^{37})$
double	64	$(-10^{37}, -10^{-37}) \cup \{0\} \cup (10^{-37}, 10^{37})$

În afara tipurilor de bază parțial analizate mai sus, în limbajul C există o clasă practic infinită de tipuri derivate construite pe baza celor fundamentale prin: *tablouri* de obiecte de un tip dat; *funcții* care returnează obiecte de un tip dat; *pointeri* spre obiecte de un tip dat; *structuri* conținînd secvențe de obiecte de diverse tipuri; *reuniuni* care pot conține obiecte de diverse tipuri.

În general, aceste metode de construire a obiectelor pot fi aplicate recursiv.

Tabelul I.1 cuprinde o situație comparativă a tipurilor de date fundamentale și derivate în principalele limbaje pentru programare de sistem.

În majoritatea cazurilor, în programe C se folosesc întregi, numere în virgulă mobilă, texte și pointeri.

Întregii — cu sau fără semn — sînt numere de forma 1, −33,25 etc. Valorile posibile depind de tipul *int* folosit pentru reprezentarea datelor. De obicei tipul *int* este un întreg care reflectă dimensiunea naturală a întregilor de pe calculatorul gazdă (nu mai puțin de 16 biți). Tipurile *short int* și *long int* (prescurtat *short* și *long*) au o lungime ce depinde de implementare. Tipurile *int*, *short int* și *long int* reprezintă valori întregi cu semn, în timp ce prefixate cu *unsigned* — valori întregi pozitive (deci mai mari sau egale cu zero).

Numerele în virgulă mobilă au parte fracționară (3,141593) și exponent ($1{,}6 \cdot 10^{-19}$). În general, ele sînt numere reale. Programele P6.C și

Scanned with OKEN Scanner

Tabelul I.1.

Tipuri de date în principalele limbaje pentru programare de sistem

Limbaje	C	Ada	Modula-2	Pascal concurent	Mesa	EDISON	CLU
Tipuri primitive							
Boolean	—	da	da	da	da	da	da
Enumerare	enum	da	da	da	da	da	—
Scalar	—	da	da	da	da	da	—
Caracter	char	da	da	da	da	da	da
Întreg	int	da	INTEGER	da	INTEGER,	da	da
	short int,		CARDINAL		LONG INTEGER,		
	long int,				CARDINAL,		
	unsigned short,				LONG		
	unsigned long				CARDINAL		
Virgulă mobilă	float,	FLOAT	—	REAL	REAL	—	REAL
	double,						
	long double						
Tipuri derivate							
Pointeri	*	ACCESS	POINTER	—	POINTER,		
					DESCRIPTOR	—	da
Articol	struct	RECORD	RECORD	— RECORD	RECORD	RECORD	RECORD, STRUCT
Tablou	da	ARRAY	ARRAY	ARRAY	ARRAY	ARRAY	SEQUENCE
Reuniune	union	da	da	—	da	—	da
Mulțime	—	—	SET	SET	—	SET	—

Scanned with OKEN Scanner

P7.C arată modul cum se lucrează cu numere reale de tip *float* sau *double*.

Textele sînt construite din caractere singulare (A, b, 5, ?) și din șiruri de caractere („Introduceți două numere:"). Caractere separate pot fi descrise prin date de tip *char*. Limbajul C nu pune la dispoziție un tip de dată special pentru șiruri de caractere. În schimb, șirurile de caractere pot fi definite prin conceptele: *vector de caractere* și *pointer-caracter*.

Folosindu-se conceptul *vector de caractere* programul P8.C poate fi rescris sub forma (P9.C):

```
/*
 *        Program P9.C
 */
main()
{
        char nume[]  = "ION ALEXANDRU";
        char adresa[]= "Str. Al. Russo,nr. 5";

        puts(nume);
        puts(adresa);
}
```

În program se declară doi vectori de caractere (tip *char*). Aici *nume*[], *adresa*[] sînt notații standard pentru astfel de vectori, în timp ce simbolurile *nume* și *adresă* reprezintă adresele în memorie ale celor doi vectori. Cei doi vectori sînt inițializați cu șirurile de caractere menționate în dreapta semnului „==". Cum fiecare caracter ocupă un bait, vectorul *nume*[] va ocupa 13, iar *adresa*[], 21 de locații memorie. În realitate primului vector i se alocă 14 locații, iar celui de-al doilea 22, deoarece la sfîrșitul fiecărui șir se generează cîte un caracter NULL (valoarea '\0'), numit și terminator de șir. De exemplu, vectorul *nume*[] va fi inițializat astfel:

I	O	N		A	L	E	X	A	N	D	R	U	/0

În acest fel, începutul șirului este referit în program prin identificatorul *nume*, iar sfîrșitul lui, prin caracterul NULL. Funcția *puts*() se folosește de aceste precizări. Astfel, *puts(nume)* primește adresa primului caracter din șir. Dacă acest caracter este diferit de caracterul '\0', atunci el este trimis spre *stdout*. Se continuă astfel pînă cînd se recunoaște terminatorul de șir. În acest moment *puts*() trimite spre dispozitivul de afișare un caracter "\n", și returnează controlul funcției

Scanned with OKEN Scanner

apelante. Un element al unui asemenea vector este un caracter. Orice caracter din vector poate fi referit însoțind numele vectorului de o valoare numerică care reprezintă poziția relativă a acestuia în cadrul vectorului. Primul caracter din șir are poziția relativă zero, al doilea 1 etc. Deci, *nume*[*0*] referă caracterul 'I', *nume*[*1*] caracterul 'O' etc. De pildă, prin secvența:

putchar(*nume*[0]);
putchar(*nume*[1]);
putchar(*nume*[2]);

se scrie *stdout* subșirul 'ION'. Dacă se dorește scrierea subșirului 'ALEXANDRU', atunci se poate folosi funcția *puts*(), dîndu-i drept argument adresa elementului 4 din vector. În C aceasta se notează cu & *nume*[*4*] sau *nume* + *4*; deci, *puts*() poate fi apelată în unul din următoarele moduri:

puts(&*nume*[4]);

sau

puts(nume + 4);

În ambele cazuri efectul este același.

A doua metodă folosită pentru definirea unui șir de caractere o oferă conceptul de *pointer-caracter*. Ea constă în păstrarea într-o zonă-memorie de un cuvînt a adresei de început a șirului de caractere, șir care uzual este terminat cu caracterul '\0'. În limbajul C o variabilă de tip adresă (sau pointer) se notează cu un identificator precedat de caracterul '*'. Astfel, notația **nume* este un pointer, iar declarația

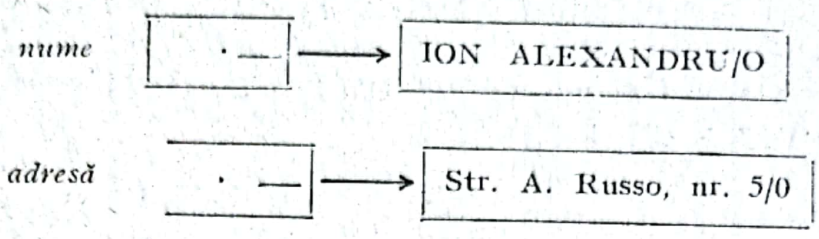
char*nume;

definește un *pointer-caracter* sau un pointer spre un șir de caractere. Cu această notație programul P9.C se poate rescrie sub forma (P10.C):

```
/*
 *      Program P10.C
 */
main()
{
        char *nume,*adresa;

        nume  = "ION ALEXANDRU";
        adresa= "Str. Al. Russo,nr. 5";

        puts(nume);
        puts(adresa);

}
```


Scanned with OKEN Scanner

În această situație valorile variabilelor *nume* și *adresa* sint adresele de început ale celor două șiruri de caractere



De obicei, un astfel de pointer ocupă un cuvînt-memorie.

3.3. Identificatori și cuvinte-cheie

În scopul referirii diverselor obiecte, în cadrul programelor C se folosesc *identificatori*. Ei se utilizează pentru a denumi variabile, constante, tipuri de date, funcții. În programele prezentate pînă aici au fost folosite asemenea elemente, de exemplu:

nume, adresă, a, b, suma — variabile definite în programe;
main — funcția principală a oricărui program C;
printf, puts, putchar, scanf — funcții din biblioteca C;
char, int, float, double — tipuri de date de bază.

Identificatori de tipul *char, int, float, double, short, long, signed, unsigned* sînt rezervați, ei folosindu-se în programe ca și *cuvinte-cheie*. Aceștia nu pot fi utilizați cu altă semnificație. Lista cuvintelor-cheie cuprinde identificatorii:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Unele implementări extind această listă și cu alți identificatori (de exemplu, *asm, fortran, pascal*).

Construirea unui identificator corect presupune respectarea următoarelor reguli:

— un identificator este o secvență de litere, cifre (0, 1, ..., 9) și semnul '—' (liniuța de subliniere);

Scanned with OKEN Scanner

— primul caracter obligatoriu trebuie să fie o literă sau liniuța de subliniere;

— în limbajul C literele mari sînt diferite de literele mici;

— identificatorii pot avea orice lungime, însă conform standardului ANSI cel puțin primele 31 caractere sînt semnificative (de exemplu, în implementarea TURBO C sînt semnificative primele 32 caractere).

3.4. Constante

În programele C constantele pot fi: întregi, în virgulă mobilă sau reale, constante-caracter, constante-șir și enumerări.

Constantele întregi se introduc prin secvențe de cifre zecimale, eventual urmate de sufixele *l*, *L*, *u*, *U*. Astfel o constantă de tipul 567 este de tip *int*, în timp ce 40 000, de tip *long*. Acesta din urmă se poate indica și prin sufixare cu *l* sau *L*. De pildă, 567*l* sau 567*L* sînt de tip *long*. În același mod se pot defini și alte tipuri (de exemplu, 40000*u* sau 40000*U* sînt de tipul *unsigned int*, în timp ce 50000*ul* sau 50000*UL* de tipul *unsigned long*).

Valorile întregi pot fi definite și ca secvențe de cifre octale (prin prefixare cu zero) sau hexazecimale (prin prefixare cu zero urmat de *x* sau *X*). În ultimul caz, cifrele de la 10 la 15 se înlocuiesc cu litere de la *a* la *f* sau *A* la *F*. Astfel, 0567 este de tip *int* și corespunde numărului octal 567, în timp ce 0567*L* este de tip *long* și corespunde aceluiași număr octal. La fel, OX56A este de tip *int* și corespunde numărului hexazecimal 56A, în vreme ce OX56A*L* este de tip *long int*.

Observație. O constantă întreagă zecimală fără sufix va fi reprezentată în ordine în unul din tipurile: *int*, *long int*, *unsigned long int*. Dacă este octală sau hexazecimală fără sufix se alege în ordine unul din următoarele moduri de reprezentare: *int*, *unsigned int*, *long int*, *unsigned long int*. Cînd sufixul este *u* sau *U*, atunci se alege tipul *unsigned int* ori *unsigned long int*, iar în cazul sufixării cu *l* sau *L* se alege în ordine *long int* sau *unsigned long int*.

Constantele în virgulă mobilă corespund numerelor reale reprezentate cu marcă zecimală (—253.73), cu exponent (5*e*—7) sau combinat (1.6*e*—19). Implicit tipul lor este *double*. Prin sufixare cu *f* sau *F* tipul devine *float*, iar cu *l* sau *L* — *long double*. De exemplu, —253.73*F* este de tip *float*, în timp ce —253.73*L*, de tipul *long double*.

Constanta-caracter este un întreg scris ca un caracter între apostrofuri. Valoarea unei constante-caracter este chiar valoarea numerică corespunzătoare caracterului dat în setul de caractere al mașinii. De pildă, dacă pe un calculator caracterele se reprezintă în cod ASCII,

Scanned with OKEN Scanner

```
/*
 *      Program P11.C
 */

#include <stdio.h>

main()
{
        char litera;

        scanf("%c",&litera);
        putchar(litera-'a'+'A');
}
```

atunci constanta '1' are valoarea 061_8, 49_{10} sau 31_{16}. Constantele-caracter pot fi folosite în operații de calcul exact ca și întregii. De exemplu, prin programul (P11.C) pentru o literă mică tastată de la terminal, se afișează litera mare corespunzătoare. Dacă litera mare tastată este *b* și pentru reprezentarea caracterelor se folosește codul ASCII, atunci expresia *(litera* — *'a'* + *'A')* este echivalentă cu expresia $152_8 - 151_8 + 101_8 = 102_8$, adică tocmai codul ASCII pentru litera *B*.

Caracterele pot fi reprezentate și prin *secvențe-escape*, (de exemplu, prin constanta '\n' se introduce caracterul *newline*). De asemenea, ele pot fi reprezentate și prin valorile octale ('\OOO' — 3 cifre octale) sau hexazecimale ('\xhh' — două cifre hexazecimale precedate de litera *x*) corespunzătoare. De exemplu, caracterul ASCII BEL poate fi introdus sub forma:

```
#define BEL '\007'
```

sau

```
#define BEL '\x07'
```

Setul complet de *secvențe-escape* cuprinde:

\a	caracterul BEL	\\	caracterul \
\b	caracterul BS	\?	caracterul ?
\f	caracterul FF	\'	caracterul '
\n	caracterul NL(LF)	\"	caracterul "
\r	caracterul CR	\000	numărul octal 000
\t	caracterul HT	\xhh	numărul hexa hh
\v	caracterul VT	\0	caracterul NUL

Constanta-șir este o secvență de zero, unu sau mai multe caractere (inclusiv caractere reprezentate prin *secvențe-escape*), delimitată prin apostrofuri duble. O constantă-șir este reprezentată ca un vector de caractere; deci, în reprezentare internă sfîrșitul șirului este delimitat printr-un bait cu valoarea zero (caracterul '\0'). Aceasta înseamnă că nu există o limită anume referitoare la lungimea unui șir de caractere, însă, pentru a o determina, un program trebuie să examineze șirul în

Scanned with OKEN Scanner

cauză. Funcția *strlen*(*s*) examinează un șir *s* și returnează lungimea lui (fără caracterul-terminator). De exemplu, după apelul:

l = *strlen* („ION ALEXANDRU");

variabila *l* va avea valoarea 13.

Un astfel de șir poate fi copiat într-un vector de caractere, folosindu-se funcția standard *strcpy*(). De exemplu, fie declarația:

char nume[31];

adică un vector-caracter ce poate fi ocupat de un șir de lungime maximă 30. Există posibilitatea de a atribui o valoare acestei variabile, utilizîndu-se funcția *strcpy*() sub forma:

strcpy(*nume*, „ION ALEXANDRU");

În acest caz din vectorul *nume*[31] se vor folosi doar primele 14 caractere.

Două constante de tip șir de caractere pot fi concatenate prin juxtapunere. De pildă, constanta-șir "ABCD" "EFGH" este echivalentă cu „ABCDEFGH". Delimitatorul de șir poate fi introdus în constantă printr-o secvență-escape. Astfel, șirul ABC''DE se exprimă ca o constantă-șir sub forma "ABC\"DE" și nu prin dublarea apostrofului ca în alte limbaje.

Constanta-enumerare — sau, simplu, *enumerarea* — este o listă de valori întregi constante. Deci, se folosește pentru a descrie o mulțime discretă de valori întregi. De exemplu, se poate utiliza declarația:

enum zile {LUNI, MARȚI, MIERCURI, JOI, VINERI, SÎMBĂTĂ, DUMINICĂ};

Lista de nume reprezintă constante întregi, prima dintre ele reprezentînd întregul zero, a doua 1, a treia 2 etc. Dacă unui nume din listă i se atribuie o valoare, atunci celor care urmează li se vor asocia valori în raport cu aceasta. De exemplu, mai corect, declarația anterioară trebuie scrisă sub forma:

enum zile {LUNI = 1, MARȚI, MIERCURI, JOI, VINERI, SÎMBĂTĂ, DUMINICĂ};

În acest fel fiecare nume va identifica corect una din cele 7 zile ale săptămînii.

Numerelor din listă li se pot atribui și valori distincte, ca în exemplul următor:

enum secv-esc {BEL='\a', BS='\b', FF='\f', NL='\n', CR='\r'};

Evident că la două nume diferite din listă li se poate asocia aceeași valoare.

Enumerarea constituie de multe ori o metodă convenabilă de a asocia valori întregi unor nume și o alternativă la declararea de constante simbolice prin directiva # *define*.

Scanned with OKEN Scanner

3.5. Declarații

Toate variabilele folosite într-un program C trebuie să fie declarate. O astfel de declarație specifică un tip și o listă de una sau mai multe variabile care trebuie să aibă tipul precizat. De exemplu, prin instrucțiunea:

```
int a, b, c, suma;
```

se declară variabilele *a*, *b*, *c* și *suma* ca fiind de tip întreg.

Prin această declarație se precizează cantitatea de memorie alocată fiecărei variabile (cîte un cuvînt în cazul de mai sus), precum și mulțimea valorilor pentru fiecare dintre acestea.

În momentul în care este declarată, o variabilă poate primi și o valoare inițială. Valoarea se specifică sub forma unei constante, compatibilă cu tipul, sau a unei *expresii constante*, adică o expresie compusă din constante, constante simbolice și operatori, expresie a cărei valoare poate fi determinată la compilare. Iată cîteva exemple

```
char null = '\0';
int delta = 'a' - 'A';
float epsilon = 1.0e-5;
char nume[ ] = „ION ALEXANDRU";
```

În cazul în care valoarea unei variabile este constantă, deci nu se schimbă pe parcursul execuției programului, atunci ea poate fi protejată folosindu-se în declarație calificatorul *const*. Exemple:

```
const float pi = 3.1415926;
const double e = 2.71828182845905;
const int minint = -32768;
char const *format = „%7.3f %15s%5d\n";
char const mesaj[] = „Atenție:";
```

În ultimul exemplu, se declară un vector-caracter cu valoare constantă, iar în penultimul un pointer spre un șir cu valoare constantă.

3.6. Operatori și expresii de asignare

Cea mai uzuală operație este asignarea, ca în expresia $suma = a + b + c$. Operatorul de asignare este semnul '='; valoarea expresiei din dreapta se atribuie variabilei din stînga operatorului '='. În C forma:

$$suma = a + b + c$$

trebuie privită ca o nouă expresie, numită *expresie de asignare*. Valoarea ei este chiar valoarea expresiei din dreapta operatorului de asignare.

Scanned with OKEN Scanner

În consecință, în aceeași instrucțiune pot fi construite asignări multiple, ca de exemplu:

$$x = y = z = 0;$$

În acest caz evaluarea se face de la dreapta spre stînga, deci mai întîi i se atribuie lui z valoarea zero, apoi valoarea expresiei de asignare $z = 0$, care este chiar zero, se atribuie lui y, după care valoarea expresiei de asignare $y = (z = 0)$, care este tot zero, i se atribuie lui x. Folosindu-se paranteze pentru a marca expresiile de asignare, instrucțiunea de mai sus s-ar scrie sub forma:

$$(x = (y = (z = 0)));$$

O expresie de asignare de forma

$$x = x + 5$$

în care variabila din stînga apare imediat în dreapta operatorului '=' se poate scrie într-o formă compactă de tipul

$$x + = 5$$

De această dată s-a folosit operatorul '+=', care este tot un operator de asignare. Această formă condensată are efecte și în codul obiect generat, în sensul că acesta este adesea mai eficient decît în cazul formei clasice. Majorității operatorilor binari le corespund operatori de asignare de forma '*op*=', unde *op* poate fi:

$$+, -, *, /, \%, \ll, \gg, \&, \hat{}, |.$$

Deci, situațiile menționate, o expresie de asignare de forma:

$$var = (var)op(expr.)$$

unde *var* este o variabilă și *expr* este o expresie, admite o reprezentare compactă sub forma:

$$var\ op = expr$$

În forma compactă *var* este evaluată o singură dată.

3.7. Operatori aritmetici

În expresii se pot folosi operatorii aritmetici binari +, —, *, /, corespunzători celor patru operații algebrice de bază, și % pentru operația *modulo*. Operatorul % se întrebuințează numai cu date de tip întreg. De pildă, expresia $a\%b$, unde a și b sînt variabile întregi, conduce la obținerea restului împărțirii lui a la b. Acest rest este zero atunci cînd b divide pe a. În aceleași condiții, expresia a/b produce cîtul împărțirii întregi a lui a la b. În expresii operatorii + și — au aceeași precedență, care însă este mai mică decît a grupului *, / și %. Precedența ultimului grup la rîndul ei este mai mică decît cea a operatorilor

unari + și —. Folosirea parantezelor în expresii poate schimba precedența între operatori în timpul evaluării acestora. *Exemple.* Dacă a, b, c, d sînt variabile de tip *int*, atunci

- expresia $d*b\%a$ este echivalentă cu $(d*b)\%a$;
- expresia $-a/d$ este echivalentă cu $(-v)/d$;
- expresia $a\%-b*c$ este echivalentă cu $(a\%(-b))*c$;
- expresia $a = b = c = d - 15$ este echivalentă cu $a = (b = = (c = (d - 15)))$.

3.8. Operatori de incrementare și decrementare

În C operațiile de forma $i = i + 1$, $j = j - 1$ pot fi programate folosindu-se doi operatori unari specifici, și anume ++ (pentru incrementare cu 1) și —— (pentru decrementare cu 1). Aspectul neobișnuit al utilizării acestor operatori constă în faptul că ei pot fi folosiți ca prefix pentru variabile (de exemplu, $++i$ și $--j$) sau ca sufix ($i++$ și $j--$). Astfel, în expresia $++i$, variabila i este incrementată înainte de a-i folosi valoarea, în timp ce în expresia $i++$, variabila i este incrementată după întrebuințarea valorii acesteia. Dacă i are valoarea 5, atunci în expresia $k = i++$ se atribuie lui k valoarea 5, după care i devine 6, iar în expresia $k = ++i$ se incrementează i (deci devine 6), iar în continuare se atribuie lui k valoarea 6. Aceste aspecte pot fi verificate, consultîndu-se programul P12.C:
Fiind operatori unari, ++ și —— au în expresii precedență mai mare decît operatorii binari. De pildă, expresia $k++/++i*--j$ este echivalentă cu $(((k++)/(++i))*(--j)$, unde k, i, j sînt variabile întregi.

```
/*
 *      Program P12.C
 */
main()

        int  i,j,s;
        char *format;

        format="i=%d,j=%d,s=%d\n";
        i=j=s=0;
        s=++i + ++j;printf(format,i,j,s);
        s=i++ + j++;printf(format,i,j,s);
        s=++i + j++;printf(format,i,j,s);
        s=i-- + --j;printf(format,i,j,s);
        s=i++ + j;  printf(format,i,j,s);
        s=++i + j;  printf(format,i,j,s);
        s=--i + j;  printf(format,i,j,s);
        s=i-- + j;  printf(format,i,j,s);
```


Scanned with OKEN Scanner

Acești doi operatori sînt foarte eficienți, însă ei trebuie utilizați cu mare atenție. Iată un exemplu: Fie $a[]$ un vector de caractere ocupat de o secvență de litere mici care trebuie transformate în litere mari. Pentru a transforma caracterul de rang i se folosesc două secvențe aparent cu același efect, și anume:

$$a[i] -= ('a' - 'A'); ++i;$$

și

$$a[++i] -= ('a' - 'A');$$

Cu a doua secvență însă modificarea se face nu pentru caracterul i, ci pentru $i + 1$. Corect este ca ea să fie scrisă sub forma:

$$a[i++] -= ('a' - 'A');$$

De asemenea, în timp ce expresiile echivalente

a[i] −= ('a' −'A'); și a[i] = a[i] − ('a' − 'A');

au efecte identice, nu același lucru se poate spune despre cazul următor:

a[i++] −= ('a' −'A'); și a[i++] = a[i++] − ('a' −'A');

Dacă în primul caz expresia a[i++] este evaluată o singură dată, în al doilea ea este evaluată de două ori.

3.9. Operatori pentru prelucrare biți

Pentru a manipula secvențe de biți, în C se folosesc următorii șase operatori: & (operatorul ȘI); | (operatorul SAU); ^ (operatorul SAU exclusiv ori sumă modulo 2); « (decalare stînga); » (decalare dreapta); ~ (complement față de unu sau negare).

Ei pot fi utilizați numai pentru date de tip *char*, *short int*, *int* și *long int* cu sau fără semn. Operatorii &, |, ^, « , » sînt binari, iar al șaselea, unar. Obiectele asupra cărora se fac operațiile sînt șiruri de biți. La nivelul fiecărui operand se operează bit cu bit în conformitate cu tabelul următor:

x	y	x&y	x|y	x^y	~x
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Operatorii de decalare ≪ și ≫ realizează deplasarea la stînga, și respectiv la dreapta a biților operandului din stînga cu un număr de poziții binare dat de valoarea operandului din dreapta fiecărui operator. Operandul al doilea trebuie să aibă o valoare întreagă pozitivă. Astfel,

Scanned with OKEN Scanner

dacă x este o variabilă de tip *int*, atunci $x \ll 2$ este echivalentă cu o înmulțire cu 4 (valoarea lui x este deplasată spre stînga cu două poziții binare, fără a afecta bitul semn, iar biții din dreapta sînt completați cu zero). Operația $x \gg 2$ este echivalentă cu o împărțire întreagă la 4 (ultimii doi biți se pierd, iar semnul se propagă spre dreapta). Dacă x este de tipul *unsigned (short, int, long* sau *char)*, atunci la decalarea spre dreapta pozițiile binare libere din stînga se completează cu zero.

Pentru observarea efectelor acestor operatori, pe propria implementare de limbaj C, se poate executa următorul program (P13.C):

```
/*
 *      Program P13.C
 */
main()
{
        unsigned int  a,b,c;
        int d,e;
        char *format1,*format2,*format3;

        format1="%6o %2s %6o = %6o\n";
        format2="%c %6o = %6o\n";
        format3="%4d %2s %4d = %4d\n";
        a=07760; b=0177400; d=-256;

        c = a & b;  printf(format1,a,"&",b,c);
        c = a | b;  printf(format1,a,"|",b,c);
        c = a ^ b;  printf(format1,a,"^",b,c);
        c = a << 4; printf(format1,a,"<<",4,c);
        c = a >> 4; printf(format1,a,">>",4,c);
        c = b << 1; printf(format1,b,"<<",1,c);
        c = b >> 1; printf(format1,b,">>",1,c);
        e = d << 2; printf(format3,d,"<<",2,e);
        e = d >> 2; printf(format3,d,">>",2,e);
        c = ~a;     printf(format2,"~",a,c);
        c = -a;     printf(format2,"-",a,c);
}
```

3.10. Operatori de tip adresă

În C, operarea cu adrese presupune folosirea următorilor doi operatori: &, semnificînd „adresa lui . . .", și *, semnificînd „adresare indirectă".

Primul operator returnează adresa unei variabile date. Dacă *a* este o variabilă de tip *int*, *float*, *double* sau *char*, atunci &*a* înseamnă adresa locației de memorie asociată variabilei *a*. La fel, dacă *nume* este un pointer spre un șir de caractere, atunci **nume* referă primul caracter din șir. Programul P14.C exemplifică aceste noțiuni.

Scanned with OKEN Scanner

```
/*
 *      Program P14.C
 */
main()
{
        int   a;
        char *nume;

        a = 1930;
        nume = "ION ALEXANDRU";
        printf(" a = %d, &a = %p\n",a,&a);
        printf(" *nume = %c,nume = %p\n",*nume,nume)
```

Cu descriptorul *%p* se afișează o adresă; în cazul de față, adresa lui *a* (adică *&a*) și adresa primului caracter din șirul adresat indirect prin pointerul *nume*.

3.11. Operații de intrare

Prin astfel de operații este posibilă introducerea de date în timpul execuției unui program de la tastatură, dintr-un fișier memorat pe disc etc. În continuare vor fi prezentate cîteva din funcțiile standard folosite în acest scop.

Funcția scanf(), echivalentul funcției *printf*() pentru operații de intrare, permite citirea cu format a unor date introduse prin dispozitivul standard de intrare, *stdin*. Se folosește în conformitate cu formatul:

scanf(⟨*șir-format*⟩, ⟨*adr1*⟩, ⟨*adr2*⟩, ...);

Prin *scanf*(), sub controlul ⟨*șirului-format*⟩, se citesc de pe dispozitivul de intrare date, care — după conversie — sînt atribuite argumentelor din listă, în ordine. Spre deosebire de *printf*(), aici argumentele trebuie furnizate prin adresele lor: ⟨*ard1*⟩, ⟨*adr2*⟩, ... Cînd explorarea *șirului-format* se încheie, citirea de date se termină, chiar dacă lista de argumente nu a fost epuizată. După execuție, funcția *scanf*() returnează o valoare întreagă (de tip *int*), reprezentînd numărul de valori citite, convertite și atribuite argumentelor din listă. Dacă citirea nu se face corect (sau se ajunge la sfîrșitul fișierului) se obține ca retur valoarea EOF. *Șirul-format* conține de obicei specificatori de conversie folosiți pentru interpretarea datelor introduse, eventual separați prin spații sau caractere HT, VT, FF, care sînt ignorate. Un *specificator de format* începe cu caracterul % și este urmat de obicei de un caracter de conversie care poate fi: *d* — pentru numere întregi în baza 10; *i* — pentru întregi în octal (încep cu zero) sau în hexazecimal (încep cu *0x* sau *0X*); *o* — pentru întregi în octal; *x* — pentru întregi în hexazecimal; *u* — pentru întregi, fără semn, în zecimal; *c* — pentru

Scanned with OKEN Scanner

caractere (inclusiv spații și tab-uri); *s* — pentru șiruri de caractere (diferite de spațiu, HT, VT, NL, CR, FF); *e, f, g* — pentru numere în virgulă mobilă; *n* — scrie în variabila asociată (de tip *int*) numărul de caractere citite prin funcția *scarf*() curentă. Litera din specificatorul de conversie poate fi precedată de caracterul '*', cînd trebuie suprimată asignarea valorii citite cu un argument din listă (se sare astfel peste un cîmp) sau de un număr ce indică dimensiunea cîmpului de citit. Un cîmp de citit este definit ca un șir ce conține caractere diferite de spațiu, HT, NL, CR, VT, FF; numite și „spații albe". El se extinde pînă la următorul „spațiu alb" sau pe lungimea indicată în specificatorul de conversie (dacă există). În general, „spațiile albe" în șirul citit sînt considerate ca separatori de cîmpuri.

Dacă *a* și *b* sînt două variabile de tip *int*, atunci pentru a citi valorile lor de pe dispozitivul *stdin* se poate folosi apelul:

scanf(„%d%d", &a, &b);

Cele două valori se introduc una după alta separate prin cel puțin un caracter de tip „spațiu alb". Dacă se dorește separarea numerelor prin virgulă, atunci se apelează funcția *scanf*() sub forma:

scanf(„%d, %d", &a, &b);

Se remarcă de asemenea utilizarea în lista de argumente a adreselor variabilelor *a* și *b*. În cazul în care o variabilă identifică un vector-caracter, în *scanf*() este suficient să se precizeze doar numele variabilei deoarece el indică chiar adresa de început a vectorului. De pildă, dacă vrem să citim *numele*, *prenumele* și *vîrsta* unei persoane în variabilele:

```
char nume[20], prenume[20];
int vîrsta;
```

atunci poate fi folosită funcția *scanf*() sub forma:

scanf(„%s%s%d", nume, prenume, &vîrsta);

Cînd cele trei date sînt înregistrate într-un fișier și nu interesează *prenumele*, funcția *scanf*() se va apela sub forma:

scanf(„%s %*s %d", nume, &vîrsta);

Prin citirea unui șir de caractere cu descriptorul %s, la sfîrșitul acestuia se adaugă caracterul terminator ('\0'). Deci, vectorul-caracter corespunzător trebuie dimensionat ca să includă și acest caracter.

Funcția gets() permite citirea unei linii introduse prin *stdin* și memorarea acesteia într-un vector-caracter, a cărui adresă este dată ca argument. În vector, caracterul terminator de linie (*newline*) este înlocuit cu caracterul „\0". Se pot citi în acest caz șiruri de caractere care cuprind și „spații albe" (mai puțin caracterul *newline*, interpretat ca terminator de linie).

Scanned with OKEN Scanner

Astfel, pentru a citi în vectorul *nume* atît numele, cît și prenumele unei persoane se poate folosi apelul

gets(*nume*);

La sfîrșit de fișier și în caz de eroare în citire, *gets*() returnează pointerul NULL (definit în ⟨*stdio.h*⟩); altfel, returnează adresa de început a vectorului unde s-a făcut citirea (un pointer). Acest lucru este ilustrat în programul (P15.C) care citește mai multe linii de pe dispozitivul *stdin* și le afișează pe *stdout*.

```
/*
 *      Program P15.C
 */
#include <stdio.h>

main()
{
        char *c,*gets();
        char linie[81];
        while((c=gets(linie)) != NULL)
                puts(c);
}
```

Se citesc linii de maximum 80 caractere. Valoarea returnată de *gets*() este atribuită variabilei *c*. Dacă valoarea expresiei de asignare *c* = *gets*(*linie*) este diferită (s-a folosit operatorul relațional '!=' care înseamnă „diferit de") de valoarea NULL înseamnă că s-a citit o linie și atunci se execută instrucțiunea *puts* (*linie*) — adică linia este scrisă pe *stdout*. Pentru execuția repetată a operațiilor s-a folosit mecanismul *while* sub forma

while (⟨*condiție*⟩)
    instrucțiune;

care înseamnă că atît timp cît condiția este îndeplinită se execută instrucțiunea care urmează. În caz contrar se trece mai departe.

Funcția getchar() se folosește pentru a citi următorul caracter de pe dispozitivul *stdin*. Caracterul citit (conform tipului *unsigned char* și convertit în *int*) este returnat de *getchar*(), deci poate fi memorat undeva sau scris într-un fișier. Dacă nu există caractere pe *stdin* sau apare o eroare, atunci *getchar*() returnează valoarea EOF. De exemplu, pentru a citi o secvență de caractere introduse prin *stdin* și a o afișa pe *stdout* este uzual programul P16.C:

Scanned with OKEN Scanner

```
/*
 *      Program P16.C
 */
#include <stdio.h>

main()
{
        int  c;

        while((c=getchar()) != EOF)
                putchar(c);
}
```

Se remarcă faptul că funcția *getchar*() nu are parametri de intrare.

3.12. Operatori relaționali

Această categorie de operatori permite compararea a două valori și luarea unei decizii, după cum rezultatul comparării este *adevărat* sau *fals*. Dacă rezultatul operației este *fals*, atunci valoarea rezultată este zero, iar dacă este *adevărat*, valoarea este unu. În C se folosesc următorii operatori relaționali: > (mai mare), < (mai mic), >= (mai mare sau egal), <= (mai mic sau egal), == (egal), != (diferit de).
În expresii, operatorii <,>, <=, > = au aceeași precedență, în timp ce == și != au o precedență inferioară.

Dacă *e1* și *e2* sînt două expresii care trebuie comparate, atunci semantica operatorilor relaționali este dată prin tabelul:

e1−*e2*	*e1*<*e2*	*e1*>*e2*	*e1*<=*e2*	*e1*>=*e2*
pozitiv	0	1	0	1
zero	0	0	1	1
negativ	1	0	1	0

Acest lucru este important de reținut. De exemplu, o expresie de forma $3 < j < 5$ acceptată în C și echivalentă cu $(3 < j) < 5$ prin evaluare poate conduce la un rezultat eronat. Dacă $j = 2$, atunci, în ordine, expresia $3 < j$ are valoarea zero (*fals*), iar $0 < 5$, valoarea 1 *adevărat*). Deci, după evaluare expresia $3 < j < 5$, pentru $j = 2$, are valoarea *adevărat*, ceea ce nu este conform cu realitatea. Corect este ca ea să fie scrisă ca o conjuncție de două condiții.

În cazul operatorilor == și !=, calculînd diferența *e1*−*e2*, rezultă numai două situații:

e1−*e2*	*e1*==*e2*	*e1*!=*e2*
zero	1	0
diferit de zero	0	1

Expresia *e1*!=*e2* este echivalentă cu expresia logică !(*e1*==*e2*). Operatorii relaționali au o precedență mai mică decît operatorii aritmetici, astfel că o expresie de forma *a*<*b*+*c* este interpretată ca *a*<(*b*+*c*), ceea ce era de așteptat.

Expresiile de asignare pot fi incluse ca termeni în expresii relaționale, însă trebuie avut în vedere că precedența operatorilor de asignare este mai mică decît a operatorilor relaționali. Astfel, în programul P16.C s-a folosit expresia relațională:

(*c* = *getchar*()) !=NULL

Primul termen este o expresie de asignare a cărei valoare logică este 1, dacă valoarea atribuită lui *c* este diferită de zero, și 0 în caz contrar. Dacă expresia s-ar scrie sub forma:

c = getchar() !=NULL

atunci ea ar fi echivalentă cu *c* = (*getchar*() !=NULL); deci *c* nu ar primi în acest caz decît valorile 0 și 1 în funcție de rezultatul evaluării expresiei dintre paranteze.

3.13. Operatori și expresii logice

Operatorii logici binari && (SI) și || (SAU), precum și operatorul logic unar de negare !, atunci cînd sînt aplicați unor expresii, conduc la valorile întregi 0 și 1, cu semnificația *fals* și *adevărat*. Semantica acestor operatori se poate deduce din tabelul următor, unde *e1* și *e2* sînt două expresii:

e1	e2	e1&&e2	e1||e2	!e1
zero	zero	0	0	1
zero	diferit de zero	0	1	1
diferit de zero	zero	0	1	0
diferit de zero	diferit de zero	1	1	0

Expresiile legate prin operatorii logici binari sînt evaluate de la stînga la dreapta; evaluarea se oprește atunci cînd rezultatul *(fals* sau *adevărat)* este deja cunoscut.

De exemplu, expresia logică

(*car*==' ')||(*car*=='\n')||(*car*=='\t')

verifică dacă un caracter (*car*) este un *spațiu alb*. Evaluarea se face de la stînga la dreapta și se oprește în momentul în care se găsește un

Scanned with OKEN Scanner

spațiu sau un *newline* ori, în fine, un *caracter HT*. Exemplele următoare arată modul în care se evaluează expresiile logice:

- expresia $i\&\&j\&\&k$ este echivalentă cu $(i\&\&j)\&\&k$;
- expresia $a\&\&b||c-5$ este echivalentă cu $(a\&\&b)||(c-5)$;
- expresia $a<b\&\&b<c$ este echivalentă cu $(a<b)\&\&(b<c)$;
- expresia $a!=b||!c+5$ este echivalentă cu $(a!=b)||((!c)+5)$.

Tabelul I.2.

Precedența și asociativitatea operatorilor în C

Operatori	Asociativitate
( ) [ ] → .	De la stînga la dreapta
! ~ ++ −− + − * & (*tip*) *sizeof*	De la dreapta la stînga
* / %	De la stînga la dreapta
+ −	De la stînga la dreapta
≪ ≫	De la stînga la dreapta
< <= > >=	De la stînga la dreapta
== !=	De la stînga la dreapta
&	De la stînga la dreapta
^	De la stînga la dreapta
|	De la stînga la dreapta
&&	De la stînga la dreapta
||	De la stînga la dreapta
?: (operator ternar)	De la dreapta la stînga
= += −= *= /= %= &= ^= |= ≪= ≫=	De la dreapta la stînga
, (operatorul virgulă)	De la stînga la dreapta

3.14. Precedența și asociativitatea operatorilor

În tabelul I.2 sînt ilustrate regulile de precedență și asociativitate pentru toți operatorii. O parte din acești operatori vor fi prezentați

Scanned with OKEN Scanner

în paragrafele următoare. Operatorii de pe aceeași linie au aceeași precedență, cu precizarea faptului că operatorii unari +, —, * au precedență mai mare decît operatorii binari similari. Pe linii precedența scade de sus în jos. De exemplu, operatorii <, <=, >, >= au aceeași precedență, care la rîndul ei este mai mare ca a operatorilor ==, != și mai mică ca a operatorilor ≪, ≫.

În practică, pentru a evita o serie de ambiguități, este recomandată utilizarea parantezelor pentru a specifica exact ordinea în care termenii expresiilor trebuie evaluați.

3.15. Instrucțiuni și blocuri

O expresie sau un apel de funcție devine instrucțiune dacă este urmată de un caracter „;". Astfel, în C caracterul „;" este un terminator pentru o instrucțiune. În contextele sintactice unde este necesară o instrucțiune, dar semantic nu se cere nici o operație, se poate folosi conceptul de *instrucțiune vidă*, scrisă ca un singur „;".

Parantezele { și } se folosesc pentru a grupa declarații și/sau instrucțiuni într-o *instrucțiune compusă* sau un *bloc*. O instrucțiune compusă este la rîndul ei o instrucțiune. Din această cauză, o instrucțiune compusă poate fi plasată în orice context sintactic în care se folosește o instrucțiune. Exemplu tipic de instrucțiune compusă este corpul oricărei funcții (inclusiv *main*()). De asemenea, instrucțiunile de tip *if*, *if-else*, *while*, *for*, *do*, *switch*, folosite pentru controlul fluxului execuției într-un program, necesită de cele mai multe ori utilizarea conceptelor de instrucțiune compusă și instrucțiune vidă.

3.16. Instrucțiunea if

O instrucțiune *if* cu care se implementează în C o structură de control de selecție sau o structură alternativă are următorul format generic:

```
if (⟨expresie⟩
        ⟨instrucțiune 1⟩
else
        ⟨instrucțiune 2⟩
```

unde ⟨*expresie*⟩ semnifică orice expresie care prin evaluare conduce la o valoare întreagă. Dacă valoarea expresiei este diferită de zero (*adevărat*), atunci se execută ⟨*instrucțiune* 1⟩; altfel, dacă valoarea este zero (*fals*), se execută ⟨*instrucțiune* 2⟩. În ambele cazuri controlul este

Scanned with OKEN Scanner

transferat mai departe la instrucțiunea ce urmează după *if*. Aici prin ⟨*instrucțiune* 1⟩ sau ⟨*instrucțiune* 2⟩ se înțelege o instrucțiune simplă, o instrucțiune compusă (un bloc) sau o instrucțiune vidă.

Porțiunea „*else* ⟨*instrucțiune* 2⟩ este opțională, în acest fel putîndu-se obține o structură de selecție cu o ramură vidă de forma:

if (⟨*expresie*⟩)
⟨*instrucțiune*⟩

Exemple:

• Următorul program (P17.C) citește două numere și afișează pe cel mai mare dintre ele:

```
/*
 *      Program P17.C
 */
#include <stdio.h>

main()
{
        int  x,y;

        printf("Introduceti doua numere intregi: ");
        scanf("%d,%d",&x,&y);
        if (x > y)
                printf("Cel mai mare este: %d\n",x);
        else
                printf("Cel mai mare este: %d\n",y);
```

• Următoarea secvență asigură pe programator că oricare ar fi valorile anterioare ale variabilelor *a* și *b*, după execuția acesteia *a* va fi întotdeauna mai mare ca *b*:

```
if (a<b){
   temp=a;
   a=b;
   b=temp;
}
```

Deoarece prin *if* se testează doar valoarea unei expresii, o codificare de forma *if*(*expresie*) — deși mai puțin clară — este mai eficientă decît *if*(*expresie* ! = 0), de exemplu. Astfel, dacă *a* este o variabilă întreagă, este mai bine să se scrie *if*(*a*) în loc de *if* (*a*!=0), *if*(!*a*) în loc de *if*(*a*==0) sau *if*(*a*—1) în loc de *if* (*a*!=1). Prin examinarea codului obiect generat se pot găsi și alte asemenea cazuri de codificări eficiente.

Scanned with OKEN Scanner

Într-o instrucțiune *if* se poate include pe o ramură o altă instrucțiune *if*. În acest fel se creează posibilitatea de a codifica structuri de selecția multiplă (între mai mult decît două alternative), folosindu-se așa-numitele perechi *else-if*. O asemenea construcție este de forma:

```
if        (⟨expresie 1⟩)
     ⟨instrucțiune 1⟩
else if   (⟨expresie 2⟩)
     ⟨instrucțiune 2⟩
else if   (⟨expresie 3⟩)
     ⟨instrucțiune 3⟩
     .
     .
     .
else if   (⟨expresie N⟩)
     ⟨instrucțiune N⟩
else
     ⟨instrucțiune (N + 1)⟩
```

În acest caz expresiile sînt evaluate în ordine; dacă una din ele are valoarea *adevărat*, atunci instrucțiunea corespunzătoare este executată și controlul este transferat la instrucțiunea următoare din program. Codul pentru fiecare alternativă poate fi format dintr-o instrucțiune simplă (inclusiv instrucțiunea vidă) sau dintr-un bloc delimitat prin { și }. Dacă nici una dintre expresii nu este adevărată, atunci se execută secvența corespunzătoare ultimei alternative introdusă prin *else* și nu prin *else-if*. Această ultimă alternativă nu este obligatorie, structura putîndu-se încheia după secvența notată cu ⟨*instrucțiune N*⟩. Următorul program (P18.C) permite contorizarea cifrelor zecimale, a literelor mari și a celorlalte caractere care apar într-un text introdus prin *stdin*.

Deoarece partea *else* dintr-o instrucțiune *if* este opțională, apare o ambiguitate atunci cînd *else* este omis dintr-un *if* inclus. Aceasta se rezolvă prin asocierea lui *else* cu cel mai apropiat *if*. De exemplu, în secvența:

```
if (a>0)
   if (x>y)
      x=a
else
   y=a;
```

deși din aliniere ar rezulta că *else* se asociază cu primul *if* în realitate ramura *else* este asociată cu al doilea *if*. Dacă într-adevăr se dorește

```
/*
 *      Program P18.C
 */
#include <stdio.h>

main()
{
        int  ncifre,nlitmin,nlitmaj,naltcar,car;

        ncifre=nlitmin=nlitmaj=naltcar=0;
        while ((car=getchar()) != EOF)
                if      ('0' <= car && car <= '9')
                        ++ncifre;
                else if ('a' <= car && car <= 'z')
                        ++nlitmin;
                else if ('A' <= car && car <= 'Z')
                        ++nlitmaj;
                else
                        ++naltcar;
        printf("Textul contine:\n"
                "%4d cifre,\n"
                "%4d litere mici,\n"
                "%4d litere mari,\n"
                "%4d alte caractere\n",ncifre,
                                       nlitmin,
                                       nlitmaj,
                                       naltcar);
```

structura dedusă din aliniere atunci secvența trebuie rescrisă sub una din formele:

a)
```
if (a>0){
   if (x>y)
      x=a;
}
else
   y=a;
```

b)
```
if (a>0)
   if (x>y)
      x=a;
   else
      ;
else
   y=a;
```

3.17. Instrucțiunea while

Această instrucțiune reprezintă mecanismul sintactic de bază pentru a programa cicluri în C. El a fost folosit în programele P15.C, P16.C și P18.C. Formatul general al acestei instrucțiuni este:

while (⟨*expresie*⟩)
⟨*instrucțiune*⟩

În timpul execuției, ⟨*expresia*⟩ este mai întîi evaluată. Valoarea ei trebuie să fie întreagă. Dacă valoarea calculată este diferită de zero (*ade-*

Scanned with OKEN Scanner

vărat), atunci ⟨*instrucțiunea*⟩ este executată. Aceasta poate fi o instrucțiune simplă (inclusiv instrucțiunea vidă) sau o instrucțiune compusă. În continuare, ⟨*expresia*⟩ este din nou evaluată și procesul se repetă. Dacă după o evaluare (inclusiv prima) rezultă valoarea zero (*fals*), atunci controlul este transferat la instrucțiunea ce urmează în program după *while*. Astfel, instrucțiunea asociată cu *while* se execută repetat cît timp valoarea expresiei este diferită de zero. Evident, există posibilitatea ca această instrucțiune să nu fie executată niciodată. De asemenea, dacă expresia nu este constituită corect se poate întîmpla să se intre într-un ciclu infinit (o eroare tipică în programarea ciclurilor).

În situația că totuși dorim să programăm un ciclu infinit, atunci se poate folosi o expresie care rămîne tot timpul adevărată. Un exemplu uzual, este următorul:

```
while (1) {
    ...
}
```

Ieșirea din ciclu în acest caz se asigură prin mecanisme de tip *break*, *go to* sau *return*.

Uneori, în cicluri *while*, este adecvată folosirea instrucțiunii vide. Astfel, secvența următoare:

```
while ((c=getchar())==' '||c=='\t')
```

permite ca dintr-un șir introdus prin *stdin* să se elimine o secvență continuă de spații și/sau caractere HT.

```
/*
 *      Program P19.C
 */
#include <stdio.h>

main()
{
        int  xi,yi,x,y;

        printf("Introduceti 2 numere pozitive:");
        scanf("%d,%d",&xi,&yi);
        x=xi; y=yi;
        while ( x != y )
                if ( x > y )
                        x -= y;
                else
                        y -= x;
        printf("Cmmdc(%d,%d)=%d\n",xi,yi,x);
}
```

Programul P19.C determină cel mai mare divizor comun pentru orice pereche x, y de numere întregi pozitive. Metoda de calcul se bazează pe faptul că:

Scanned with OKEN Scanner

- dacă $x>y$, atunci $cmmdc\ (x, y) = cmmdc\ (x - y, y)$;
- dacă $x<y$, atunci $cmmdc\ (x, y) = cmmdc\ (x, y - x)$;
- dacă $x=y$, atunci $cmmdc\ (x, y) = x = y$.

De exemplu, $cmmdc\ (14, 21) = 7$.

3.18. Instrucțiunea for

Instrucțiuni de tip *for* pentru programare cicluri se găsesc în cele mai multe din limbajele de programare. Totuși, forma în care acestea apar în C este foarte flexibilă, puternică și eficientă. Utilizarea ei este legată de parcurgerea unor structuri de date de tip *tablou*, însă aici se poate folosi și în multe alte cazuri de programare. De exemplu, următorul program (P20.C) poate fi utilizat pentru a calcula, după un algoritm iterativ, valoarea $n! = 1.2.3 \ldots n$:

```
/*
 *      Program P20.C
 */
#include <stdio.h>

main()
{
        int     n,i;
        long int factorial;

        printf("Introduceti n:");
        scanf("%d",&n);
        factorial = 1;
        for ( i=1 ; i<=n ; i++)
                factorial *= i;
        printf("%d! = %ld\n",n,factorial);
}
```

Instrucțiunea *for* se folosește după următorul format general:

for (⟨*expr1*⟩ ; ⟨*expr2*⟩ ; ⟨*expr3*⟩)
⟨*instrucțiune*⟩

Această notație este echivalentă semantic cu secvența:

⟨*expr 1*⟩ ;
while (⟨*expr 2*⟩){
⟨*instrucțiune*⟩
⟨*expr 3*⟩ ;
}

Scanned with OKEN Scanner

Din punct de vedere sintactic, componentele ⟨*expr 1*⟩, ⟨*expr 2*⟩, ⟨*expr 3*⟩ sînt expresii, iar ⟨*instrucțiune*⟩, o instrucțiune simplă, o instrucțiune compusă sau, ca și în cazul lui *while*, instrucțiunea vidă. De obicei ⟨*expr 1*⟩ se întrebuințează pentru inițializarea ciclului, din care cauză este evaluată prima. Se evaluează apoi ⟨*expr 2*⟩ care trebuie să conducă la o valoare întreagă. Dacă valoarea acestei expresii este zero *(fals)* atunci execuția ciclului se termină imediat, controlul fiind transferat la următoarea instrucțiune din program. În cazul în care valoarea expresiei ⟨*expr 2*⟩ este diferită de zero *(adevărat)*, se execută ⟨*instrucțiune*⟩, se evaluează expresia ⟨*expr 3*⟩ și se revine la evaluarea expresiei ⟨*expr 2*⟩ care controlează numărul de iterații.

Oricare din cele trei expresii din *for* poate lipsi. De exemplu, programul P20.C poate fi rescris sub forma P21.C în care ⟨*expr 3*⟩ este omisă din *for*:

```
/*
 *      Program P21.C
 */
#include <stdio.h>

main()
{
        int n,i;
        long factorial;

        printf("Introduceti n:");
        scanf("%d",&n);
        factorial = 1;
        for ( i=1 ; i<=n ; )
                factorial *= i++;
        printf("%d! = %ld\n",n,factorial);
}
```

În cazul în care ⟨*expr 2*⟩ se omite din *for*, se consideră că aceasta are în permanență valoarea *adevărat*. În acest fel se programează un ciclu *for* infinit, din care se poate ieși prin mecanisme de tip *break*, *return* sau *goto*. Cel mai frecvent, un ciclu *for* infinit se programează sub forma:

```
for(;;){
   ...
}
```

Instrucțiunile *for* pot fi incluse una în alta. De exemplu, programul P22.C parcurge un șir de caractere de la stînga la dreapta, afișînd subșirurile ce au ca bază primul caracter:

Scanned with OKEN Scanner

```
/*
 *      Program P22.C
 */
#include <stdio.h>

main()
{
        int  l,n,i;
        char sir[81];

        puts("Tastati un sir terminat cu <CR>:");
        gets(sir);
        l=strlen(sir);
        for ( n=0 ; n <= l ; ++n ){
                for ( i=0 ; i < n ; ++i )
                        putchar(sir[i]);
                putchar('\n');
```

Programul P23.C realizează același lucru, dar luînd ca bază pentru subșiruri ultimul caracter din șirul inițial:

```
/*
 *      Program P23.C
 */
#include <stdio.h>

main()
{
        int  k,n,i;
        char sir[81];

        puts("Tastati un sir terminat cu <CR>:");
        gets(sir);
        n=strlen(sir);
        for ( k = n-1 ; k >= 0 ; --k ){
                for ( i=0 ; i < k ; ++i )
                        putchar(' ');
                puts(&sir[k]);
```

3.19. Operatorul „virgulă"

Operatorul binar „ ," își găsește cel mai des utilitatea în instrucțiuni *for*. El leagă două expresii sub forma:

⟨*expresie 1*⟩, ⟨*expresie 2*⟩

Se evaluează mai întîi ⟨*expresia 1*⟩, apoi ⟨*expresia 2*⟩. Expresia pe ansamblu primește valoarea și tipul operandului din dreapta opera-

Scanned with OKEN Scanner

torului *virgulă*. De exemplu, dacă i, j și k sînt variabile de tip *int* și $k = 5$, atunci expresia:

$$(i = 1,\ j = 2,\ ++k + 1)$$

va avea după evaluare tipul *int* și valoarea 7, în concordanță cu expresia $++k + 1$.

Dintre toți operatorii din C, virgula are precedența cea mai mică. Însă nu în toate cazurile ea este interpretată ca operator. De exemplu, în apelurile de funcții virgula separă argumentele în listă. Dacă operatorul virgulă trebuie folosit în acest context, atunci expresia în care acesta apare este necesar să fie inclusă între paranteze. De pildă, pentru a afișa valoarea funcției $f(x) = (x^2 + x)/(x - 5)$, cînd $x = -3$, se poate folosi instrucțiunea:

```
printf("f(x)=%f pentru x=%f\n", (x=-3.,(x*x+x)/(x-5.)),x);
```

S-a presupus că x este de tip *float* sau *double*.

În cicluri *for*, operatorul virgula este util atunci cînd același vector (sau tablou) trebuie parcurs simultan după mai mulți indici care evoluează separat. De exemplu, programul P24.C citește un șir de caractere și-l afișează inversat:

```
/*
 *      Program P24.C
 */
#include <stdio.h>

main()
{
        int  i,j;
        char ct,sir[81];

        puts("Tastati un sir terminat cu <CR>:");
        gets(sir);
        for ( i=0,j=strlen(sir)-1 ; i < j ; ++i,--j )
                ct=sir[i],sir[i]=sir[j],sir[j]=ct;
        puts(sir);
```

3.20. Instrucțiunea do-while

Spre deosebire de ciclurile programate cu *while* sau *for*, unde condiția este verificată la început, în cazul folosirii mecanismului *do-while* condiția se evaluează după execuția secvenței de instrucțiuni ce reprezintă corpul ciclului. Sintactic, această structură de control este de forma

do
⟨*instrucțiune*⟩
while (⟨*expresie*⟩);

Scanned with OKEN Scanner

Semantic *do-while* este echivalentă cu secvența:

⟨*instrucțiune*⟩
while (⟨*expresie*⟩)
⟨*instrucțiune*⟩

Se poate ușor remarca faptul că ⟨*instrucțiunea*⟩ ce constituie corpul ciclului (adică o instrucțiune simplă, o instrucțiune compusă sau instrucțiunea vidă) este executată cel puțin o dată. Celelalte execuții sînt condiționate de valoarea întreagă rezultată din evaluarea expresiei. Dacă această valoare este zero *(fals)*, atunci controlul se transferă la următoarea instrucțiune din program; în caz contrar, se execută corpul ciclului și se reevaluează expresia. Un caz tipic de utilizare a mecanismului *do-while* este oferit de programele interactive în care selecția unei opțiuni se face pe baza unui meniu afișat pe ecranul terminalului. De asemenea, este utilă pentru a verifica introducerea unor valori corecte de la tastatură. De exemplu, următoarea secvență asigură preluarea corectă a unei valori întregi între 1 și 10:

```
do {
   printf("\n\n Introduceți un întreg între 1 și 10:");
   scanf("%d",&nr);
}
while (nr<1||nr>10);
```

În tabelul I.3 se prezintă o situație comparativă privind instrucțiunile disponibile pentru controlul execuției în principalele limbaje utilizate în scopul dezvoltării de programe pentru sistem.

CAPITOLUL 4

4. PROGRAMARE AVANSATĂ ÎN C

4.1. Funcții C

Principalul mijloc prin care se pot modulariza programele C este oferit de conceptul de *funcție* (unele funcții standard au fost deja folosite pentru diverse operații). În C, orice funcție „întoarce" după apel o valoare, al cărei tip trebuie cunoscut. În practică însă de multe ori valorile returnate de funcții sînt ignorate. Standardul de limbaj C permite chiar declararea explicită a funcțiilor care nu returnează valori ca

Scanned with OKEN Scanner

Tabelul I.3.

Structuri de control în principalele limbaje pentru programare de sistem

Limbaje	C	Ada	Modula-2	Pascal concurent	Mesa	EDISON CLU	
Instrucțiuni simple							
Instrucțiune vidă,	;	NULL	empty	empty	NULL	SKIP	empty
asignare	=, +=, −=, *=, /=, %=, ...	:=	:=	:=	←	:=	:=
salt	goto	GOTO	—	—	GOTO	—	—
apel subprogram	da	da	da	da	da	da	da
Instrucțiuni de selecție							
	if,	IF THEN	IF THEN	IF THEN	IF THEN	IF	IF THEN
	if-else,	ELSIF ELSE,	ELSIF, ELSE,	ELSE,	ELSE,		ELSEIF,
	switch	CASE IS	CASE OF	CASE OF	SELECT FROM		TAGCASE
Instrucțiuni de ciclare							
	while,	LOOP,	LOOP,	FOR DO,	DO,	DO,	FOR DO,
	for,	FOR LOOP,	FOR DO,	WHILE DO,	FOR DO,		WHILE DO;
	do-while,	WHILE LOOP,	WHILE DO,	REPEAT UNTIL,	THROUGH DO,		WHILE
			REPEAT UNTIL,		WHILE DO,		
					UNTIL DO,		
	break, continue	EXIT WHEN	EXIT	CYCLE	EXIT	EXIT	BREAK, CONTINUE
Tratare excepții	—	da	—	—	da	—	da

Scanned with OKEN Scanner

fiind de tip *void*. Tipul unei funcții corespunde tipului valorii returnate. Acesta poate fi un tip de bază *(char, int, float, double* etc.*)* sau un tip derivat *(pointer, structură* etc.*)*. În plus, o funcție are argumente care descriu tipul datelor transmise în momentul apelării acesteia. Pot exista și funcții fără argumente. De exemplu, funcția *getchar*() este apelată fără argumente.

În C o funcție poate fi *definită*, *declarată* și *apelată*. Atunci cînd o funcție se definește i se dă efectiv codul C din care este compusă. Conform standardului de C acest cod se organizează după formatul:

```
⟨tip⟩ ⟨nume-funcție⟩ (⟨declarații argumente⟩
{
    ⟨declarații-locale⟩
    ⟨instrucțiuni⟩
}
```

deși este acceptată și o formă mai veche de definire

```
⟨tip⟩ ⟨nume-funcție⟩ (⟨listă-argumente⟩)
⟨declarații-argumente⟩
{
    ⟨declarații-locale⟩
    ⟨instrucțiuni⟩
}
```

De exemplu, funcția *max(a,b)*, care returnează cel mai mare dintre numerele întregi *a* și *b*, se poate defini sub forma (P25.C):

```
/*
 *      Program P25.C
 */

int max(int a,int b)
{
        if (a > b)
                return (a);
        else
                return (b);
}
```

Precizăm că instrucțiunea *return* este folosită pentru a returna dintr-o funcție, spre funcția apelantă, o valoare care corespunde tipului funcției curente. În general ea se folosește cu formatele:

return(⟨*expresie*⟩); sau *return*⟨*expresie*⟩;

Scanned with OKEN Scanner

unde ⟨*expresie*⟩ este o expresie a cărei valoare (după o conversie adecvată) este returnată funcției apelante. Conversia ține seama de tipul declarat pentru valoarea returnată de funcție.

Apelul unei funcții înseamnă referirea funcției, împreună cu valorile actuale pentru argumente, precum și preluarea valorii returnate dacă este necesar. Astfel, în cazul funcției definite mai sus apelul poate fi, de exemplu, de forma:

$$x = max(i + 2,5);$$

Pentru a genera corect codul obiect, o funcție trebuie declarată în funcția care o apelează sau în afara acesteia. În acest fel în textul sursă curent se va ști exact tipul valorii returnate (implicit se presupune tipul *int*) precum și tipul argumentelor cu care funcția trebuie apelată. De pildă, dacă vrem să apelăm din funcția *main*() pe *max*(), atunci codul sursă poate arăta sub forma P26.C:

```
/*
 *      program P26.C
 */
#include <stdio.h>
int _Cdecl max(int a,int b);
main()
{
        int x,y;
        printf("Tastati 2 numere intregi: ");
        scanf("%d,%d",&x,&y);
        printf("Cel mai mare este: %d\n",max(x,y));
}
```

Acum, funcția *max*() se poate compila separat sau împreună cu funcția apelantă. În cazul în care declararea se face sub forma

$$int\ max(\ );$$

tipurile argumentelor cu care funcția este apelată nu mai pot fi verificate.

În cadrul unei aplicații, declarațiile de funcții apelate pot fi adunate într-un fișier prefix și introduse în programe, folosind directiva ***#include***.

În C transmiterea parametrilor de la funcția apelantă spre funcția apelată se face *prin valori*. Dacă este necesară transmiterea parametrilor *prin adrese*, atunci în apel în loc de valori se folosesc adrese, iar în definiție parametrii formali se declară ca pointeri. De exemplu, o funcție care schimbă valorile a două variabile reale se poate defini sub forma P27.C:

Scanned with OKEN Scanner

```
/*
 *      functia schimba()   (P27.C)
 */

void schimba(float *x,float *y)
{
        float temp;
        temp = *x;
        *x   = *y;
        *y   = temp;
}
```

Programul P28.C arată modul de apel al acestei funcții:

```
/*
 *      Program P28.C
 */
#include <stdio.h>

void schimba(float *x,float *y);
main()
{
        float x,y;
        scanf("%f,%f",&x,&y);
        schimba(&x,&y);
        printf("x= %f,y=%f\n",x,y);
}
```

Se remarcă folosirea adreselor variabilelor *x* și *y* în apelul funcției *schimbă*().

4.2. Variabile externe și variabile interne

Drept *variabile interne* se consideră argumentele și variabilele definite în interiorul funcțiilor. Aceste obiecte nu sînt cunoscute decît în cadrul funcțiilor respective. La rîndul lor, *variabilele externe* sînt definite în exteriorul oricărei funcții (inclusiv *main*()) și potențial ele sînt disponibile spre a fi utilizate de toate funcțiile unui program.

Funcțiile sînt întotdeauna obiecte externe, deoarece în C nu este posibilă definirea de funcții noi în interiorul altor funcții.

Folosindu-se variabile externe, două sau mai multe funcții dintr-un program pot comunica între ele prin date comune. Astfel, orice funcție poate referi potențial o variabilă externă dacă cunoaște numele acesteia. În acest scop funcția în cauză trebuie să conțină o declarație a acestei date prefixată cu cuvîntul cheie *extern*. Fie, de exemplu programul P29.C:

Scanned with OKEN Scanner

```
/*
 *      Program P29.C
 */
#include <stdio.h>
char linie[81];

main()
{
        CitesteLinie();
        ScrieLinie();
}
```

Pentru a avea acces la variabila externă *linie*, funcțiile apelate trebuie scrise sub forma P30.C:

```
/*
 *      Program P30.C
 */

CitesteLinie()
{
        extern char linie[];
        gets(linie);
}

ScrieLinie()
{
        extern char linie[];
        puts(linie);
}
```

Astfel construite, aceste două funcții pot fi compilate separat, ele putînd comunica prin variabila externă *linie* definită în funcția *main*(). Dacă cele două funcții se compilează împreună cu funcția *main*(), codul lor sursă fiind inserat după cel al funcției principale, atunci declarațiile de tip *extern* nu mai sînt necesare, variabila externă *linie* fiind cunoscută în toate funcțiile.

Cînd funcțiile unui program partajează un număr mai mare de date comune, definirea lor sub formă de variabile externe constituie o cale mai eficientă și mai convenabilă decît listele lungi de argumente pentru funcții. Totuși, această metodă trebuie utilizată cu multă grijă, deoarece ea are efecte negative asupra structurii și calității programelor. Ea conduce la programe cu multe conexiuni între module. Se ajunge astfel la un *cuplaj* între module *prin date externe* și implicit la module cu *consistență comunicațională*, *procedurală*, *clasică*, *logică* sau *întîmplătoare* [11], de calitate mai slabă decît cea a modulelor cu *consistență funcțională*. *Cuplajul prin date externe* are următoarele conse-

Scanned with OKEN Scanner

cințe: independența mică a modulelor, susceptibilitate mare de erori, reutilizabilitatea redusă a modulelor în alte programe și extensibilitatea mică a programelor.

Variabilele definite în cadrul unei funcții nu numai că sînt *locale*, deci cunoscute în interiorul funcției gazdă, dar în mod normal ele există numai pe durata execuției acesteia. Fiecare variabilă internă dintr-o funcție este creată (i se alocă spațiu, i se atribuie o valoare etc.) atunci cînd funcția este apelată și dispare cînd funcția își termină execuția. Din această cauză acestea sînt cunoscute ca *variabile automate* sau *variabile dinamice*. Deci, variabilele dinamice nu pot păstra valori de la un apel la altul. În schimb, variabilele externe au spațiu alocat permanent și pot păstra valori pe întreaga durată a execuției programului. Aceste variabile sînt permanente.

Variabilele interne pot deveni și ele permanente dacă sînt declarate ca *variabile statice*. În aceste condiții, ele pot păstra valori de la o execuție la alta a funcției gazdă. Pentru aceasta în declarare li se adaugă prefixul *static*. De exemplu, prin declarația:

static int a, b, c;

în interiorul unei funcții, variabilele *a, b, c* devin variabile permanente cunoscute local.

Declarația *static* aplicată variabilelor externe are ca efect limitarea domeniului în care acestea sînt cunoscute doar la restul textului sursă curent ce trebuie compilat.

Astfel, dacă în exemplul anterior se dorește ca variabila *linie* să fie cunoscută doar în cadrul funcțiilor *CitesteLinie*() și *ScrieLinie*(), nu și în funcția *main*(), atunci se construiește un singur fișier cu codul sursă al celor două funcții sub forma P31.C:

```
/*
 *      Program P31.C
 */

static char linie[81];

CitesteLinie()
{
        extern char linie[];
        gets(linie);
}

ScrieLinie()
{
        extern char linie[];
        puts(linie);
}
```


Scanned with OKEN Scanner

Se obține un modul cu *consistență informațională*, care execută două operații asupra unei variabile ce este cunoscută numai în acest modul. Ea este complet ascunsă pentru restul programului care încorporează acest modul. În acest mod se pot condifica în C module bazate pe *principiul ascunderii informației* (sau *principiul lui Parnas*).

Prefixul *static* se poate folosi și în declarații de funcții. Ca și în cazul variabilelor externe și aici declarația *static* determină ca funcția în cauză să fie cunoscută numai în restul fișierului sursă de compilat.

Variabilelor dinamice și parametrilor formali din oricare funcție li se poate adăuga în declarație prefixul register. Unei astfel de variabile prin generarea de cod se caută să i se aloce spațiu în unul din registrele calculatorului gazdă. Această stare nu o pot avea decît variabile de tip *int*, *long* sau *char*. Iată cîteva astfel de declarații:

```
register int i;
register char c;
```

Plasînd în registrele calculatorului unele variabile frecvent folosite, se obțin programe mai mici și mai eficiente în execuție.

4.3. Funcții recursive

Funcțiile C pot fi recursive adică se pot autoapela direct sau indirect. De exemplu, este cunoscut faptul că $n!=n.(n-1)!$. Pe baza acestei relații de recurență se poate scrie o funcție C recursivă care să determine valoarea lui $n!$, n fiind transmis ca parametru. Dacă se mai are în vedere faptul că $0!=1$, atunci această funcție poate fi P32.C:

```
/*
 *      Program P32.C
 */

long factorial(register int n)
{
        if (n==0)
                return(1);
        else
                return(n*factorial(n-1));
}
```

sau în notația veche, P33.C:

Atunci cînd o funcție se autoapelează recursiv, la fiecare invocare se memorează pe stivă atît valorile parametrilor actuali, cît și întregul set de variabile dinamice definite în cadrul funcției, astfel că în acest caz o dimensionare corectă a stivei programului poate fi necesară.

Scanned with OKEN Scanner

```
/*
 *      Program P33.C
 */

long factorial(register int n)
{
        if (!n)
                return(1);
        else
                return(n*factorial(n-1));
}
```

Un alt exemplu interesant este oferit de șirul lui Fibonacci, în care termenul general a_n este dat de relația de recurență $a_n = a_{n-1} + a_{n-2}$, unde $a_0 = 0$ și $a_1 = 1$. Codul funcției poate fi (P34.C):

```
/*
 *      Program P34.C
 */

long fib(int n)
{
        if (n==0)
                return(0);
        else if (n==1)
                return(1);
        else
                return(fib(n-1)+fib(n-2));
}
```

Utilizarea recursivității poate să nu conducă la o reducere a memoriei necesare, atît timp cît stiva este folosită intens pentru fiecare apel recursiv. De asemenea, nici execuția poate să nu fie mai rapidă. Dar codul recursiv este mai compact și adesea mai ușor de scris și înțeles decît echivalentul său nerecursiv. Recursivitatea este convenabilă în mod deosebit pentru operații pe structuri de date definite recursiv, cum sînt listele, arborii etc.

4.4. Expresii condiționale

O astfel de expresie este scrisă folosindu-se operatorul ternar „?:" și constituie o modalitate de a include în expresii structuri de selecție binară (tip *if*). O asemenea expresie se compune după formatul general:

$$\langle expr1 \rangle ? \langle expr2 \rangle : < \langle expr3 \rangle$$

Aici expresia $\langle expr1 \rangle$ este evaluată prima. Dacă valoarea ei este *adevărat*, atunci se evaluează expresia $\langle expr2 \rangle$, rezultatul obținut con-

stituind și valoarea întregii expresii condiționale. Dacă ⟨*expr1*⟩ are valoarea *fals*, atunci se evaluează ⟨*expr3*⟩ și rezultatul obținut reprezintă chiar valoarea expresiei condiționale.

Cu aceste precizări, funcția *max*() definită în §4.1 se poate rescrie sub forma P35.C:

```
/*
 *      Program P35.C
 */

int max(int a,int b)
{
        return((a>b)?a:b);
}
```

Valoarea unei expresii condiționale poate fi atribuită unei variabile. De exemplu, dacă se intenționează definirea unei funcții *max3*() care să determine cel mai mare dintre trei numere întregi transmise ca parametri, atunci se poate folosi codul din P36.C:

```
/*
 *      Program P36.C
 */

int max3(int a,int b,int c)
{
        int m;
        return((m=((a>b)?a:b)>c)?m:c);
}
```

În acest fel, codul sursă al programelor poate deveni mai compact, iar termenii unor expresii pot fi condiționați.

4.5. Selecție multiplă folosind instrucțiunea *switch*

Se consideră că dispunem de un program care afișează pe ecran o listă de opțiuni (un meniu), din care utilizatorul alege cîte una și o execută. Pentru fiecare opțiune selectată se apelează o funcție ce execută operația cerută. Programul P37.C poate fi un exemplu, funcțiile apelate nefiind detaliate aici.

Dacă variabila după care se face selecția este de tip *char* sau *int*, atunci aceasta se poate codifica mai elegant în C folosindu-se instrucțiunea *switch* și nu *else if*. Secvența P38.C arată acest lucru:.

Scanned with OKEN Scanner

```
/*
 *      Program P37.C
 */
#include <stdio.h>

main()
{
        char c[2];
        do {
                printf("Lista optiuni de lucru:\n"
                       "   A - adaugare\n"
                       "   M - modificare\n"
                       "   S - stergere\n"
                       "   H - help\n"
                       "   T - Terminare lucru\n");
                scanf("%1s",c);
                if      (c[0]=='A') adauga();
                else if (c[0]=='M') modifica();
                else if (c[0]=='S') sterge();
                else if (c[0]=='H') help();
                else if (c[0]=='T') ;
                else                printf("Raspuns eronat\n");
        }
        while ( c[0] != 'T' );


/*
 *      Program P38.C
 */
#include <stdio.h>

main()
{
        char c[2];
        do {
                printf("Lista optiuni de lucru:\n"
                       "   A - adaugare\n"
                       "   M - modificare\n"
                       "   S - stergere\n"
                       "   H - help\n"
                       "   T - Terminare lucru\n");
                scanf("%1s",c);


                switch(c[0]){
                        case 'A': adauga();   break;
                        case 'M': modifica(); break;
                        case 'S': sterge();   break;
                        case 'H': help();     break;
                        case 'T':             break;
                        default : printf("Raspuns eronat\n");


        while ( c[0] != 'T' );
```


Scanned with OKEN Scanner

Selecţia unui caz se face în funcţie de valorile variabilei *c*, care aici este de tip *char*. Dacă valoarea lui *c* coincide cu una din cele menţionate se execută secvenţa de instrucţiuni asociată cazului selectat. Instrucţiunea *break* care încheie o secvenţă selectată permite transferul controlului la instrucţiunea ce urmează în program imediat după *switch*. Dacă o secvenţă nu conţine o instrucţiune *break*, atunci se execută instrucţiunile secvenţei următoare şi se continuă astfel pînă cînd se întîlneşte o astfel de instrucţiune sau se ajunge la sfîrşitul instrucţiunii *switch*. În cazul în care valoarea variabilei de selecţie nu coincide cu nici una din cele menţionate se execută secvenţa definită pentru cazul *default*. Dacă acest caz nu există (el este opţional), atunci instrucţiunea *switch* transferă pur şi simplu controlul mai departe. Două sau mai multe situaţii pot fi tratate prin aceeaşi secvenţă. De exemplu, dacă în cazul anterior se acceptă ca răspuns şi litere mari şi litere mici, atunci selecţia multiplă se poate rescrie sub forma P39.C.

```
/*
 *      Program P39.C
 */
#include <stdio.h>

main()

        char c[2];
        do {
                printf("Lista optiuni de lucru:\n"
                       "  A - adaugare\n"
                       "  M - modificare\n"
                       "  S - stergere\n"
                       "  H - help\n"
                       "  T - Terminare lucru\n");
                scanf("%1s",c);


                switch(c[0]){
                        case 'a':
                        case 'A': adauga();   break;
                        case 'm':
                        case 'M': modifica(); break;
                        case 's':
                        case 'S': sterge();   break;
                        case 'h':
                        case 'H': help();     break;
                        case 't':
                        case 'T':             break;
                        default : printf("Raspuns eronat\n");


        while ( c[0] != 'T' && c[0] != 't');
```


Scanned with OKEN Scanner

Lista valorilor după care se face selecția diverselor cazuri se poate defini și prin constante de tip *enum*.

4.6. Instrucțiunile *break* și *continue*

În afară de cazul menționat mai sus (instrucțiunea *switch*), instrucțiunea *break* se poate executa și în cadrul unui ciclu programat cu *while*, *do* sau *for*. În această situație execuția instrucțiunii *break* implică terminarea imediată a iterației curente (în punctul în care s-a executat *break*) și transferul controlului la instrucțiunea ce urmează imediat după cea cu care s-a programat ciclul. Schema generală de utilizare este de forma

```
while (<expresie>)  {
    . . . .
    if (<condiție>) break;
    . . . .
}
```

Astfel, dacă în una din iterații ⟨*condiția*⟩ din *if* este îndeplinită, atunci ciclul se termină automat, altfel el poate continua, pînă cînd ⟨*expresie*⟩ are valoarea *fals*. În particular, instrucțiunea *break* permite ieșirea din cicluri infinite, ca în cazul

```
for(;;) {
    . . .
    if(expresie) break;
    . . .
}
```

Dacă instrucțiunea *break* se execută în cadrul unei instrucțiuni *switch*, care la rîndul ei este inclusă într-un ciclu programat cu *while*, *for*, *do-while*, atunci ea determină doar terminarea instrucțiunii *switch* nu și ieșirea din ciclu.

La rîndul ei, instrucțiunea *continue* executată într-un ciclu implică oprirea iterației curente și asigură trecerea imediată la iterația următoare. În acest fel se poate optimiza viteza de execuție a ciclurilor.

4.7. Pointeri

Un *pointer* este o variabilă care păstrează adresa unei date și nu valoarea acesteia. În acest fel, ea poate fi referită direct prin adresa ei în memorie (dacă este cunoscută) sau indirect prin intermediul unui pointer care conține adresa ei. Utilizarea pointerilor are avantajul că prin schimbarea adreselor ce le conțin se pot manipula informații din diverse locații din memorie. De exemplu, se pot parcurge în acest fel

Scanned with OKEN Scanner

elementele unei liste. Prima dată pointerul va referi primul element din listă. Dacă fiecare element din listă conține adresa elementului următor, atunci prin actualizarea valorii unui pointer se pot parcurge toate elementele acesteia. Pointerii pot fi de asemenea utilizați în *alocarea dinamică* de spațiu pentru unele *obiecte* (variabile, tablouri, structuri etc.). O funcție care alocă dinamic spațiu, de obicei returnează adresa zonei-memorie alocată. Această adresă se poate memora într-un pointer și astfel obiectul în cauză poate fi referit.

O *variabilă pointer* se definește cu operatorul unar *. Astfel, declarația

*int *pa;*

definește un pointer ce referă o variabilă întreagă (mai exact de tip *int*). Declarația

*float *px;*

definește un pointer ce va conține adresa unei variabile de tip *float*.

Fie următoarele declarații:

```
int a = 5;
int b = 10;
int *p1, *p2;
```

atunci

`p1 = &a;`	inițializează pointerul *p1* cu adresa variabilei *a*;
`p2 = &b;`	inițializează pointerul *p2* cu adresa lui *b*;
`b = *p1;`	atribuie lui *b* valoarea 5;
`*p1 = 0;`	atribuie lui *a* valoarea zero;
`*p2 = *p1;`	atribuie lui *b* valoarea zero;
`*p1 = *p1 + 10;`	incrementează *a* cu 10;
`b = *p1 + 1;`	atribuie lui *b* valoarea lui *a* mărită cu 1;
`*p1 += 1;`	incrementează *a* cu 1;
`++*p1;`	incrementează *a* cu 1;
`(*p1)++;`	incrementează *a* cu 1;
`*p1++;`	incrementează pointerul *p1* cu 2 deoarece *p1* adresează o variabilă de tip *int*;
`*(p1 + 1) = 20;`	memorează valoarea 20 în locația de adresă *p1* + 2; dacă *p1* referă adresa *a* atunci memorarea se va face la adresa *a* + 2;
`p1 = p2;`	copie conținutul variabilei *p2* în *p1*, deci *p1* și *p2* vor referi variabila *b*.

Dacă *p* este o variabilă de tip pointer declarată generic prin

*⟨tip⟩ *p;*

Scanned with OKEN Scanner

atunci o expresie de forma $p + i$ desemnează adresa-memorie $p+i$ *$sizeof(\langle tip \rangle)$. Operatorul *sizeof* ($\langle tip \rangle$) „întoarce" lungimea spațiului-memorie (în baiți) alocat pentru o variabilă de tipul desemnat. Astfel, *sizeof(int)* este 2, *sizeof(float)* este 4 etc. Deci, dacă *px* este un pointer de tip *int*, atunci prin instrucțiunea $*(px+i)=10$ se va memora valoarea 10 în cuvîntul de adresă $px+i*sizeof(int) = px+i*2$.

4.8. Vectori de pointeri

Deoarece pointerii sînt variabile, din ei se pot construi vectori ca și în cazul celorlalte variabile. Acești vectori în care elementele sînt pointeri poartă numele de *vectori de pointeri*.

De pildă, dacă prin declarația

int a[10];

se declară un vector de întregi cu 10 componente, atunci prin declarația:

int **a*[10];

se definește un vector de pointeri cu 10 componente, fiecare pointer din vector referind o variabilă de tip *int*. Vectorii de pointeri se dovedesc foarte utili, atunci cînd referă date de tip șir de caractere. Aceste date sînt în general de lungime variabilă, astfel că folosirea vectorilor de pointeri permite memorarea compactă a acestora.

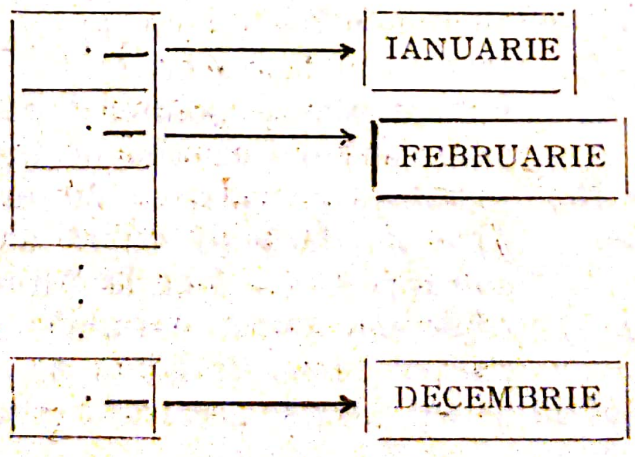
De exemplu, dacă prin declarația

char luna [ ] = „IANUARIE";

se definește un vector a cărui valoare este un șir de caractere (cîte un caracter pe componentă), prin declarația

```
char * luna [ ] = { "IANUARIE", "FEBRUARIE", "MARTIE",
                    "APRILIE", "MAI", "IUNIE", "IULIE",
                    "AUGUST", SEPTEMBRIE", "OCTOM-
                    BRIE", „NOIEMBRIE", „DECEMBRIE"};
```

se definește și se inițializează un vector de pointeri (cu 12 componente), fiecare componentă constituind adresa unui șir de caractere reprezentînd numele uneia din cele 12 luni ale anului sub forma:



Vectori de pointeri — Șiruri de caractere

4.9. Funcţii care returnează pointeri

Unele funcţii din C în loc să returneze valori, returnează pointeri sau adrese. Aceste adrese pot referi date de tip *int*, *float*, *double*, *char*, *struct* etc. De exemplu, dacă funcţia:

int f( );

returnează un întreg de tip *int*, atunci funcţia

*int *f( );*

returnează un pointer către o dată de tip *int*. Un caz interesant este oferit de funcţiile ce returnează pointeri către şiruri de caractere. O astfel de funcţie se declară sub forma:

*char *f( );*

În programul P40.C se arată modul în care se defineşte, se declară şi se apelează o astfel de funcţie. Este vorba de funcţia *NumeLuna( )* care — primind ca argument o valoare întreagă între 1 şi 12 — returnează un pointer spre un şir de caractere ce reprezintă numele uneia din cele 12 luni ale anului.

```
/*
 *      Program P40.C
 */
#include <stdio.h>

main()
{
        int i;
        extern char *NumeLuna();

        for (i=0;i<=13;i++)
                printf("%s\n",NumeLuna(i));
}

char *NumeLuna(nr)
int nr;
{
        static char *luna[]={"Eroare",
                             "Ianuarie","Februarie",
                             "Martie","Aprilie","Mai",
                             "Iunie","Iulie","August",
                             "Septembrie","Octombrie",
                             "Noiembrie","Decembrie"};

        return((nr>=1)&&(nr<=12) ? luna[nr] : luna[0]);
}
```


Scanned with OKEN Scanner

BIBLIOGRAFIE

1. KERNIGHAN, B.W.; RITCHIE D.M., *THE C PROGRAMMING LANGUAGE*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1978.

2. KERNIGHAN, B.W.; RITCHIE, D.M., *THE C PROGRAMMING LANGUAGE*, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1988.

3. KELLEY, AL.; POHL, IRA, *A BOOK ON C*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park California, 1984.

4. HANCOCK, Les; KRIEGER, M., *THE C PRIMER*, McGraw-Hill Book Company, 1986.

5. *** *MICROSOFT C COMPILER*, Reference Manual, 1983.

6. *** *TURBO C, USER'S GUIDE*, Borland International, Inc., 1987.

7. *** *TURBO C', REFERENCE GUIDE*, Borland International, Inc., 1987.

8. GORLEN, K.E. *AN OBJECT-ORIENTED CLASS LIBRARY FOR C++ PROGRAMS*, Software-Practice and Experience, 1987, 17(12), p. 899—922.

9. ANDERSON, B. *TYPE SYNTAX IN THE LANGUAGE C*, ACM Sigplan Notices, 1980, 15(3), p. 21—27.

10. APPELBE, W.F.,; HANSEN, K., *A SURVEY OF SYSTEMS PROGRAMMING LANGUAGES: CONCEPTS AND FACILITIES*, Software-Practice and Experience, 1985, 15(2), p. 169—190.

11. VĂDUVA, I.; BALTAC, V.; FLORESCU, V.; FLORICICĂ, I.; JITARU, M., *INGINERIA PROGRAMĂRII*, Vol. 1, 2, Editura Academiei R. S. România, București, 1985.

12. WORTMAN, L.A.; SIDEBOTTOM, T.O, *THE C PROGRAMMING TUTOR*, Robert J. Brady Co., 1983.

13. CONROY, D.G., *DECUS, C COMPILER*, Reference Manual, 1981.

14. MINOW, M., *DECUS, C LANGUAGE SYSTEM*, Compiler & Library; Software Support Manual 1981.

15. DENNY, R., B., *DECUS C LANGUAGE SYSTEM*, RSX-11M v3.2 Executive; C Extension Library, 1982.

16. LEICHTER, J., *DECUS C LANGUAGE SYSTEM*, Utility Library; Reference Manual, 1981.

Scanned with OKEN Scanner

PARTEA a II-a

PROGRAMARE ÎN MODULA-2

CAPITOLUL 1

LEXICUL LIMBAJULUI MODULA-2

1.1. Noțiuni de modularizare cu MODULA-2

MODULA-2 a fost dezvoltat începînd cu anul 1978 de către o echipă de informaticieni ai Institutului de Tehnologie din Zürich, sub coordonarea lui Niklaus Wirth. El este o îmbunătățire a limbajului PASCAL și conține în plus tehnici de modularizare a programelor dar și noi tipuri de date.

MODULA-2, considerat de foarte mulți specialiști în ingineria programării ca drumul cel mai scurt dintre limbajele PASCAL, și ADA constituie un exemplu de îmbinare armonioasă a principiilor programării structurate cu principiile modularizării, de unde a derivat și numele limbajului*.

Principiile **programării structurate** se concretizează prin faptul că toți algoritmii de programare se pot exprima prin structuri de control ce conțin operații repetitive (de exemplu, LOOP, FOR, REPEAT), blocuri de decizie de tip IF, construcții BEGIN-END sau PROCEDURES, instrucțiunile de tip GOTO fiind absente.

Principiile modularizării se materializează prin partiționarea programului în module care interacționează între ele; datele sînt definite la nivel de modul, iar modulele se modifică și se reînlocuiesc pe principiul compilării separate.

---

* În comparație cu limbajul ADA, considerat în unanimitate limbajul anilor '80, limbajul MODULA-2 este mult mai ușor de învățat, aproximativ o săptămînă pentru cunoscătorii limbajului PASCAL.

În limbajul MODULA-2 se poate formula o gamă variată de probleme, începînd cu cele exprimate în termenii unor modele matematice sau de altă natură, pînă la cele de elaborare de sisteme software.

Entitățile limbajului MODULA-2 sînt *programul*, *modulul*, *procedura* și *blocul*. Vom insista asupra noțiunii de modul, celelalte entități fiind similare sintactic și funcțional cu termenii corespunzători din limbajul PASCAL cu excepția faptului că programul este și el de fapt un modul.

Modulele limbajului studiat sînt constituite din colecții de date, blocuri de instrucțiuni și proceduri. Modulele permit accesul la zone de date și instrucțiuni ale altor module, devenind astfel unele module *consumatoare*, altele module *producătoare*, iar o a treia categorie — module *consumatoare* și *producătoare* în același timp.

Modulele consumatoare anunță elementele pe care le utilizează din alte module prin intermediul unor liste declarate la începutul modulului, și numite **liste de import.** Similar, modulele producătoare anunță elementele pe care le pot utiliza alte module prin intermediul unei liste declarate la începutul modulului, numită **lista de export.** (Sugestiv, exportul este „o portiță" în „zidul" care ascunde celelalte informații ale modulului)*. Modulele care sînt în același timp consumatoare și producătoare posedă liste de import și cîte o listă de export.

Programul poate să nu conțină alte module, dar să le utilizeze total sau parțial prin liste de import. De asemenea, orice modul poate conține alte module.

În procesul de programare modulele pot fi clasificate astfel:

- **modulul-program** — componenta centrală a programului — este constituit din colecții de declarații (de constante, variabile, tipuri de date, proceduri și module locale) și dintr-o secvență de instrucțiuni care este de fapt secvența sintetică a programului;
- **module globale** — entități independente ce nu sînt conținute în alte module, dar implementează servicii (declarații de constante, variabile, proceduri) pentru modulul-program sau alte module globale;
- **module locale** — entități ce pot fi conținute în oricare modul, inclusiv în cele locale. Modulele locale pot și ele la rîndul lor să conțină declarații (pentru constante, variabile, proceduri, module locale), precum și secvențe de instrucțiuni.

Se poate scrie un program format dintr-un singur modul, caz în care programul devine asemănător unui program PASCAL.

---

* Ideile de modularizare a programelor s-au concretizat o dată cu lucrările lui David Parnas [11], în care se propunea ca programele mari să fie compuse din părți (module) relativ independente, iar aceste părți să poată avea acces din unul în altul numai la anumite zone de date sau instrucțiuni. Anumite zone ale unui modul devin vizibile, iar altele sînt ascunse; ele sînt proprii numai părților (modulelor) în care sînt definite, protejîndu-le astfel de orice eventual acces din alt modul. De altfel, conceptul de vizibilitate și cel complementar de ascundere a informației se află la baza programării modulare.

Scanned with OKEN Scanner

Cel mai simplu program în MODULA-2 este:

MODULE *PROG*;
END *PROG*.

El conține modulul program *PROG* și este alcătuit din două declarații:

— declarația inițială obligatorie, „**MODULE**", urmată de numele programului și semnul „;".

— declarația de sfîrșit, „**END**", urmată de numele programului și semnul „.".

Un modul-program conține linii-sursă, care exprimă algoritmul de rezolvare a unei probleme. În funcție de stilul programatorului, o parte din secvențele și datele folosite pot fi grupate în module globale, iar altă parte să rămînă în modulul program.

Modulul-program începe cu cuvîntul cheie „MODULE", urmat de numele programului, eventual prioritatea, și de punct și virgulă.

Liniile imediat următoare conțin listele de „IMPORT", după care urmează blocul de identificatori ai modulului-program și corpul modulului-program. Programul se termină cu cuvîntul cheie „END", urmat de numele programului și de semnul „.".

Sintaxa secvențelor de fraze ale unui modul program este:

ModulProgram::=
MODULE *Identificator* [*Prioritate*]„;"
[*ListadeImport*]
[*Bloc Identificatori*]
[**BEGIN**
CorpModulProgram]
END *Identificator* „."

Identificator este numele programului.

Prioritatea este un număr sau valoarea unei expresii care indică prioritatea de execuție a modulului program.

Lista de Import este enumerarea procedurilor, constantelor, variabilelor, tipurilor de date care sînt utilizate în cadrul modulului-program și sînt definite în alte module.

Bloc Identificatori conține construcții sintactice de identificatori (declarații de constante, variabile, tipuri de date, proceduri și module locale).

CorpModulProgram este o secvență de instrucțiuni cuprinsă între cuvintele rezervate „**BEGIN**" și „**END**". *Blocul* de identificatori și *corpul* modulului constituie *blocul* modulului.

Scanned with OKEN Scanner

Partiționarea programului în module are două consecințe imediate:

— posibilitatea unei ierarhizări a importanței și a *vizibilității* unor informații din cadrul programului;

— modificarea elementelor unui modul din care se efectuează importuri fără *recompilarea* obligatorie a modulului *importator*.

Deoarece toate modulele (locale, globale, program) pot conține module locale, trebuie precizat că modulele locale stabilesc domenii de vizibilitate (accesibilitate) a informațiilor folosite în program (dar nu și durata acestora) prin specificarea listelor de import și export.

Întrucît modulele globale sînt externe modulului program, interfața lor este deosebit de importantă în procesul de modularizare a programului. De aceea, un modul global este constituit din două secțiuni: de definire (**DEFINITION**), care administrează interfața modulului, și secțiunea de implementare (**IMPLEMENTATION**), ce efectuează activitățile din cadrul modulului. Facilitățile cele mai importante oferite de limbajul MODULA-2, cum ar fi operațiile de intrare-ieșire funcțiile matematice, interfețele cu sistemul de operare, operațiile cu sisteme de fișiere, suportul de programare sistem, suportul de prelucrări grafice etc., sînt realizate de module globale puse la dispoziție de mediul de programare al limbajului MODULA-2 (vezi anexa II.2).

Legăturile dintre module sînt obținute prin funcții de „**IMPORT**" și de „**EXPORT**". Funcția de „**EXPORT**" permite unui modul să specifice obiectele interne ale modulului care sînt accesibile altor module, iar cea de „**IMPORT**" dă posibilitatea unui modul să anunțe componentele externe pe care modulul dorește să le utilizeze. Toate tipurile de module (locale, globale sau program) pot importa obiecte din alte module. Exportul este de două feluri — **implicit** și, respectiv **explicit.**

Pentru exportul **implicit** toate componentele listate în secțiunea de definire (**DEFINITION**) a unui modul global sînt automat exportate. În cazul în care toate detaliile unui obiect (procedură, tip de date, variabilă sau constantă) sînt prezente în secțiunea de definire, exportul acestuia este **transparent.** Dacă o parte din detalii apar numai în secțiunea de implementare, atunci exportul este **opac.**

Exportul **explicit** — limitat la module locale — arată că numai obiectele menționate în lista de export pot fi utilizate de alte module. Nu este permis exportul în modulul program (pentru că nu are sens) sau în secțiunea de implementare (**IMPLEMENTATION**) (căci ascunde toate informațiile cuprinse în ea) a unui modul global.

În MODULA-2 programatorul poate importa sau exporta proceduri **PROCEDURE**), tipuri (**TYPE**), variabile (**VAR**) sau constante (**CONST**) prin specificarea lor în liste de import sau de export.

Scanned with OKEN Scanner

Sintaxa unei liste de export este:

ListaExport::=**EXPORT** [**QUALIFIED**] *ListaIdentificatori* " ;"

ListaIdentificatori conține numele de proceduri, tipuri, variabile sau constante care pot fi exportate.

Cuvîntul cheie **QUALIFIED** este facultativ; prezența lui obligă referirea elementelor exportate cu indicatori de componente selectate. (În blocul modulelor importatoare, pentru a ști la care din obiectele omonime se face referirea, se prefixează numele obiectului cu numele modulului exportator. Se evită astfel posibilitatea de confuzie între două obiecte cu același nume, dar provenind din module diferite.)

Sintaxa unei liste de import este:

ListaImport::=[**FROM** *Identificator*] **IMPORT** *ListaIdentificatori*" ;"

Identificator este numele modulului din care se efectuează importul.

ListaIdentificatori este lista numelor de proceduri, tipuri, variabile sau constante care vor fi importate din modulul specificat.

De menționat că lista de identificatori poate să conțină și nume de module, dar în acest caz toate elementele din lista de export a fiecăruia din aceste module vor fi automat importate. Dacă se dorește ca numai o parte din elementele modulului care efectuează exportul să fie importate, atunci se folosește opțiunea FROM *Identificator*, iar în *ListaIdentificatori* vor fi specificate numai cele dorite. Pentru identificarea lor explicită, în blocul modulelor importatoare se vor folosi componente selectate (calificatori).

1.2. Exemplu de modularizare a unui program

Prezentăm partiționarea în module a unui program și interacțiunea între acestea. Se presupune că programul prelucrează expresii algebrice și deci va efectua citiri și editări, nalize sintactice, atribuiri, calcule.

a Modulul global *Instrucțiuni* produce procedurile *instrucțiune*, *expresie*, *termen*, *factor* și *Citnumar*. El importă procedurile *Setvaloare Citvaloare*, *următor*, *caracter*, *anterior* și exportă doar procedura *instrucțiune*. Modulul global *CalculVariabile* importă și exportă proceduri, dar mai are și rolul de a ascunde structura de date care conține valorile variabilelor din expresie. El conține modulul local *Test*, care produce procedura *test* de verificare a corectitudinii numelor de variabile. Modulul global *CalcInput* exportă procedurile *următor*, *caracter*, *anterior* și ascunde structuri de caractere necesare editării. Modulul-program denumit *Calcul* importă procedurile *instrucțiune* din modulul global *Instrucțiuni* și procedurile *anterior* și *următor* din modulul global *CalcInput*, realizînd analiza expresiei și tipărirea rezultatului.

Toate modulele amintite importă proceduri din modulul global *InOut* specializat în operații de intrare-ieșire, pus la dispoziția mediului de programare MODULA-2. Citățm cîteva din procedurile lui *InOut*:

WriteString emite un șir de caractere la terminal, *WriteLn* avansează o linie la ieșire, *Read* preia un caracter de la terminal, *WriteInt* și *ReadInt* afișează și respectiv preiau numere întregi (vezi anexa II.1).

Scanned with OKEN Scanner

În figura II.1 sînt prezentate intuitiv aceste module precum și interacția lor prin export și import.

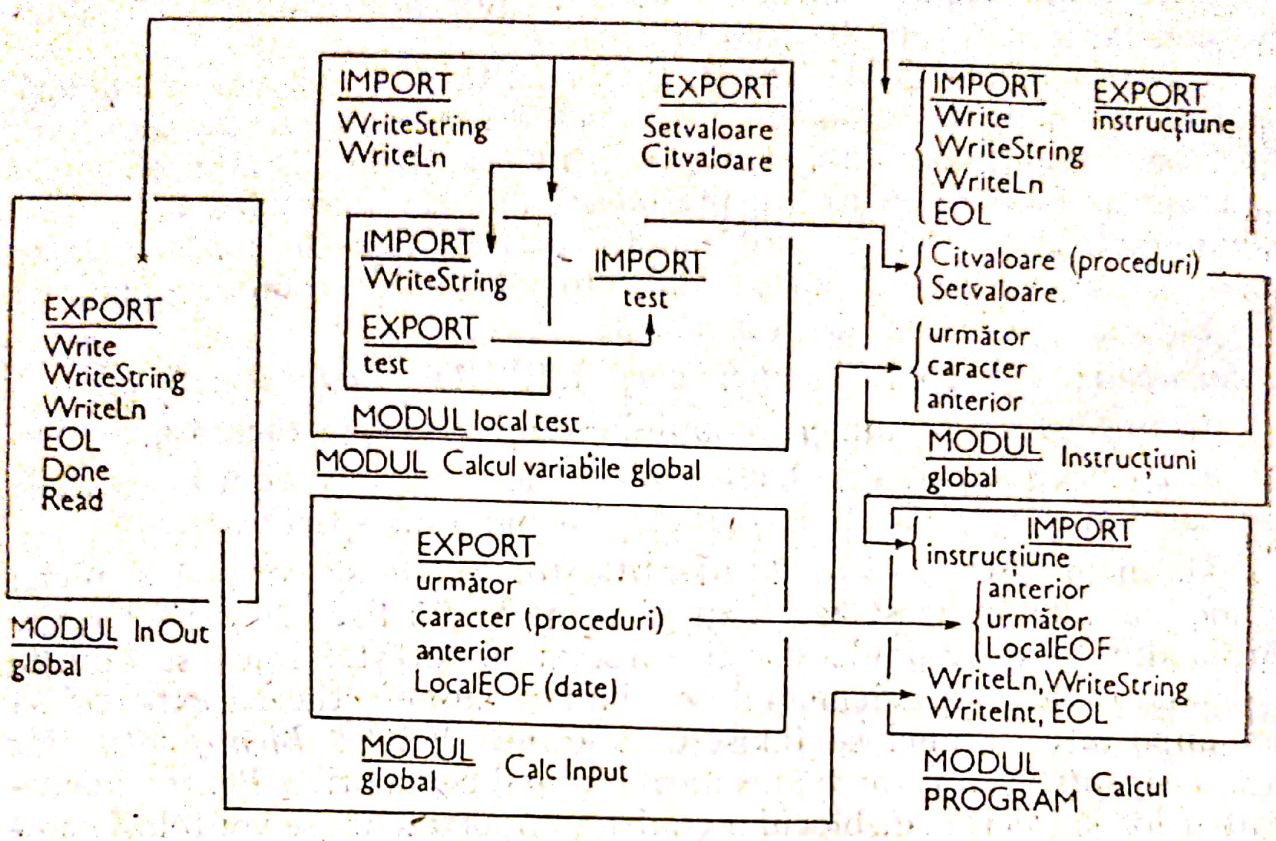


Fig II.1. Partiționarea în module a unui program pentru calculul expresiilor algebrice

Schițăm în continuare conținutul modulului global *CalculVariabile*:

```
DEFINITION MODULE CalculVariabile;
    (*secțiunea de definire a modulului global CalculVariabile*)
    (*prin prezența lor în secțiunea de definire procedurile
      SetValoare și CitValoare sînt implicit exportate*)
PROCEDURE SetValoare (W : CHAR; x : INTEGER);
    (*antetprocedură atribuire de valori variabilelor*)
PROCEDURE CitValoare (W : CHAR) : INTEGER;
    (*comentariu:antet procedură citirea valorilor variabilelor*)
END CalculVariabile.
IMPLEMENTATION MODULE CalculVariabile;
    (*secțiunea de implementare a modulului global CalculVariabile*)
FROM InQut IMPORT WriteString, WriteLn;
    (*între lista de import și corpul modulului CalculVariabilei este blocul de identificatori
      ai modulului*)
VAR
    variabile : ARRAY ["A" ... "Z"] OF INTEGER;
    (*variabilele notate de la A la Z de tip INTEGER nu pot fi folosite de alte module
      pentru că sînt definite în secțiunea de implementare*)
PROCEDURE SetValoare (W: CHAR) : INTEGER;
BEGIN
    (*corpul procedurii*)
END SetValoare;
```


Scanned with OKEN Scanner

```
MODULE Test;
      (*modulul local în care se verifică numele variabilei*)
FROM InOut IMPORT WriteString;
EXPORT test (index:CHAR): CHAR;
     (*între listele de import și export și cuvîntul BEGIN
       urmează blocul de identificatori ai modulului Test*)
PROCEDURE test (index:CHAR): CHAR;
BEGIN
     (*corpul procedurii*)
END test.
BEGIN
     (*corpul modulului local Test*)
END Test;
VAR i: CHAR;
BEGIN
     (*corpul modulului CalculVariabile*)
     FOR i:= „A" TO „Z" DO variabile [i] :=0 END;
END CalculVariabile.
```

Întrucît modularizarea este eficientă în cazul programelor cu un anumit grad de complexitate, iar în acest moment prezentarea tuturor modulelor nu este justificată, ne vom rezuma la redarea modulului-program căruia îi revine o sarcină mult simplificată.

```
MODULE Calcul;
FROM InOut IMPORT WriteString, WriteLn, WriteInt, EOL;
FROM CalcInput IMPORT următor, anterior, LocalEOL;
FROM Instrucțiuni IMPORT instrucțiune;
      (*blocul de identificatori ai programului conține o declarație de variabilă*(
      VAR
            Ch: CHAR;
      BEGIN
            (*corpul modulului program delimitat de BEGIN și END Calcul*)
      LOOP
      WriteInt (instrucțiune () , 6); WriteLn;
      Ch: = următor ()
          IF Ch =LocalEOF THEN RETURN END;
          IF Ch =EOL THEN
                WriteString („calculul expresiei");
          ELSE
                anterior (ch)
          END; (*IF*)
       END (*LOOP*)
END Calcul.
```

1.3. Elementele unui program MODULA-2

Un program în MODULA -2 este de fapt o secvență de cuvinte sau simboluri.

Simbolurile de bază se pot împărți în următoarele clase identificatori; numere; șiruri; cuvinte rezervate și operatori; comentarii; spații.

Identificatorii sînt în general nume inventate de programator (există însă și identificatori standard ai limbajului).

Scanned with OKEN Scanner

Un identificator valid reprezintă o secvență de litere mici (de la *a* la *z*), litere mari (de la *A* la Z) și cifre (de la 0 la 9), din care primul caracter este obligatoriu o literă. Sintaxa unui identificator este:

Identificator:: = *literă* {*literă* | *cifră*}

Următoarele cuvinte sînt identificatori:

Dan, xy10, yNewton, cireașă,

pe cînd următoarele nu sînt identificatori:

**, trei-cireșe, $patru, 5prune

deoarece conțin caractere diferite de litere sau cifre, iar ultimul este incorect întrucît primul caracter este cifră. Spre deosebire de limbajul PASCAL, în MODULA-2 literele mici sînt considerate diferite de literele mari de exemplu, AbC ≠ abC ≠ aBc, deoarece A ≠ a, B ≠ b, C ≠ c).

Uneori este posibil ca un identificator să provină (semantic) din alt identificator; indicatorul astfel obținut — denumit componentă selectată a primului identificator (calificator) — are sintaxa:

Calificator:: = *Identificator* {*Indentificator*}.

În afara identificatorilor folosiți de utilizator, mai există identificatori predefiniți, care depind de implementarea limbajului; de exemplu:

ABS	**CHR**	**HIGH**	**LONGREAL**	**PROC**
BITSET	**DEC**	**INC**	**MAX**	**REAL**
BOOLEAN	**EXCL**	**INCL**	**MIN**	**SIZE**
CAR	**FALSE**	**INTEGER**	**NIL**	**TRUE**
CARDINAL	**FLOAT**	**LONGCARD**	**ODD**	**TRUNC**
CHAR	**HALT**	**LONGINT**	**ORD**	**VAL**

Numerele sînt de fapt secvențe de cifre care nu trebuie să conțină spații. Numerele reale conțin punct zecimal, o parte fracționară și eventual un factor de scalare. Acesta din urmă este specificat de litera E și un întreg care este precedat sau nu de semn.

Regula de formare a numerelor este dată de sintaxa:

număr:: = *întreg* | *real*
întreg:: = *cifră* {*cifră*} [B | H]
real:: = *cifră* {*cifră*} · {*cifră*} [*factordescalare*]
factordescalare:: = E[+ | —] *cifră* {*cifră*}.

Caracterul B semnifică cifre octale, iar caracterul H cifre hexazecimale. Vom da exemple de formare a numerelor în paragraful ce descrie constantele limbajului.

Scanned with OKEN Scanner

Şirurile sînt secvenţe de caractere cuprinse între apostrofuri sau ghilimele. Astfel, dacă un şir conţine apostrofuri el trebuie închis între ghilimele şi reciproc. În acest cadru, într-un şir nu sînt permise în acelaşi timp o pereche de apostrofuri şi una de ghilimele.

Sintaxa unui şir este:

Sir::= '{*caracter*}' | "{*caracter*}".

Exemple de şiruri:

'apostrof'
"ghilimele"
'acest şir conţine:" = ghilimele'

Deoarece există un set de **cuvinte rezervate şi operatori** care nu pot fi folosite ca identificatori este necesar să fie reţinute:

AND	**ELSIF**	**LOOP**	**REPEAT**	+ =
ARRAY	**END**	**MOD**	**RETURN**	− #
				* <
				/ >
BEGIN	**EXIT**	**MODULE**	**SET**	:= <>
BY	**EXPORT**	**NOT**	**THEN**	& ⇐
				, .
CASE	**FOR**	**OF**	**TO**	; :
				( )
CONST	**FROM**	**OR**	**TYPE**	[ ]
DEFINITION	**IF**	**POINTER TO**	**UNTIL**	{ }
				^ |
DIV	**IMPLEMENTATION**	**PROCEDURE**	**VAR**	(* *)
DO	**IMPORT**	**QUALIFIED**	**WHILE**	^ |
ELSE	**IN**	**RECORD**	**WITH**	~ ,.:;...

Comentariile sînt secvenţe arbitrare de caractere cuprinse între construcţia „(*" şi construcţia „*)".

CAPITOLUL 2

VARIABILE, EXPRESII ŞI INSTRUCŢIUNI

2.1. Declaraţii

Important mecanism de definire a proprietăţilor datelor, procedurilor sau modulelor, declaraţiile indică modul prin care pot fi furnizate nume pentru obiectele declarate, natura lor şi uneori cum vor fi folosite în cadrul programului. Din aceste motive, modulele şi procedurile sînt formate din două părţi distincte:

Scanned with OKEN Scanner

• partea declarativă, ce conține declarații (numită blocul de identificatori);

• partea ce conține instrucțiuni, numită corpul modulului sau procedurii (partea cuprinsă între **BEGIN** și **END**);

Declarațiile precizează datele ce vor fi utilizate în modul sau procedură, dar au și posibilitatea de a crea noi tipuri de date, proceduri și module. Sintaxa unei declarații este:

Declarație::= *DeclarațieConstantă*|*DeclarațieVariabilă*|
DeclarațieTip|*DeclarațieProcedură*|*DeclarațieModul*

2.2. Constante

Constantele, utilizate în toate limbajele, pot servi la inițializarea variabilelor cu o valoare cunoscută, indicarea numărului de ori pentru executarea unei operații etc. În cadrul programului apar sub două forme: implicit, fără vreo precizare suplimentară, și explicit sau nominalizat, prin utilizarea declarațiilor.

Modul explicit are două avantaje: o bună alegere a numelui de constantă, ceea ce permite o mai mare lizibilitate în cadrul programului; posibilitatea de actualizare a valorii unei anumite constante, prin schimbarea unei singure declarații.

Sintaxa declarației unei constante este:

DeclarațieConstantă::= **CONST** {*identificator*:= *Expresieconstantă*";"}

MODULA-2 admite constante din tipurile: scalare; (CHAR, REAL, INTEGER, enumerative BOOLEAN, SET, etc.) Înregistrările, tablourile și tipurile de acces nu admit constante.

Exemplu:

```
MODULE Const;
    CONST
    Nzile = 7;
    Nluni = 12;
BEGIN
    (*Instrucțiuni pentru aflarea datei curente*)
END Const.
```

Forma implicită de prezentare a constantelor de tip INTEGER REAL mai este numită și forma numerică. Constantele numerice apar ca numere întregi și, respectiv, numere reale.

Numerele întregi pot fi reprezentate:

— zecimal, ca secvențe de cifre cuprinse între 0 și 9 (de exemplu, 729, 720, 30, 69 910, —7 235);

— octal, ca secvențe de cifre de la 0 la 7, urmate de sufixul 'B' (de exemplu, 325B, 62 107B, 30B, 311B);

Scanned with OKEN Scanner

— hexazecimal, ca secvențe de cifre de la 0 la 9 și litere de la A la F, urmate de sufixul 'H'.

Numerele reale conțin parte zecimală și eventual factor exponent desemnat de prefixul "E", codificate în memoria calculatorului în virgulă flotantă (de exemplu, 2.786E—2; 0.567E+4).

2.3. Variabile

Cu excepția variabilelor alocate dinamic (vezi capitolul 7), toate variabilele trebuie declarate. O declarație de variabile este o listă de identificatori urmată de numele tipului de date căruia îi aparține variabila. Sintaxa este:

DeclarațieVariabilă::= **VAR**{*ListăIdentificatori*:*Tip*;}

Vom exemplifica cu următorul modul-program ce declară variabilele *i*, *j* întregi, ARIA —număr real și *P* întreg.

```
MODULE    VarExp;
VAR
          i,j:INTEGER;
          ARIA:REAL;
          P:INTEGER;
BEGIN
     (*secvență de instrucțiuni*)
END       VarExp.
```

Ca și la declarațiile de constante, menționăm că declarațiile de variabile se poziționează în cadrul programului sau al unui modul după listele de „**IMPORT**" și „**EXPORT**" (cînd acestea există). În general există posibilitatea declarării unei variabile în blocul de identificatori al unei proceduri sau al unui modul, precum și în secțiunile de definire și implementare ale unui modul global.

2.4. Expresii

Sintactic, o expresie este o succesiune de operanzi și operatori. Evaluarea ei constă în efectuarea operațiilor într-o anumită ordine, dată de grupele de precedență, ale operațiilor. Pentru fiecare grupă de precedență ordinea operațiilor este de la stînga la dreapta. Operanzii pot fi constante, variabile sau funcții, iar după felul operatorilor putem avea expresii aritmetice, expresii logice sau expresii combinate. Operatorii au semnificații în funcție de natura operanzilor. De exemplu,

Scanned with OKEN Scanner

„+" înseamnă adunare pentru operanzi aritmetici sau logici, dar poate avea semnificația de reuniune dacă operanzii sînt de tip mulțime. Pe de altă parte, nu sînt corecte expresiile în care corespunzător unui operator operanzii sînt de tipuri diferite. Operatorii pot fi grupați prioritar în următoarele grupe de precedență:

— operatorul NOT;
— operatorii de multiplicare (*, /, DIV, MOD, AND, &);
— operatorii aditivi +, —, OR;
— operatorii relaționali =, #, <, <>, =, ⇐, >, IN.

Ca și în alte limbaje semnul — (minus) are un tratament aparte conform următoarelor reguli:

• operatorul minus este permis numai la începutul expresiilor (de exemplu, $-25+x+y$); expresia $x-25+y$ nu are sens;

• operatorul minus este permis ca urmare a unui operator de comparare (de exemplu, $x <> -25$);

• în toate celelalte cazuri operatorul unar minus și termenul următor trebuie închiși între paranteze. Exemplu: $y*(-25)+3$**MOD**$(-y)$.

Pentru exemplificare vor fi studiate cîteva cazuri de precedență: expresia $x<y<z$ nu este corectă, deoarece — datorită relațiilor de precedență — ea semnifică $(x<y)<z$, ceea ce nu are sens, întrucît $(x<y)$ este o expresie care are o valoare booleană și nu poate fi comparată cu o valoare numerică; (**NOT**x) $= y$ este incorectă; în mod corect se scrie **NOT** $(x=y)$, ceea ce este echivalent cu $x<>y$.

2.5. Asignări

Există două metode de a face ca o variabilă să aibă o anumită valoare. Metoda directă este de a utiliza o instrucțiune de asignare prin care o valoare este stabilită de către o expresie și această valoare se atribuie unei variabile. Altă metodă constă în considerarea unei variabile ca parametru într-o procedură. Acest aspect va fi detaliat în capitolul 5.

Sintaxa instrucțiunii de asignare este:

asignare::= *indicator*:= *expresie* ";"

unde

indicator reprezintă un identificator sau o construcție sintactică provenită din identificatori și caractere auxiliare ce indică o componentă selectată, un element al unui tablou, un cîmp al unei înregistrări sau o variabilă referită de pointeri.

Scanned with OKEN Scanner

Acțiunea corespunzătoare instrucțiunii de asignare constă din trei etape: evaluarea indicatorului ce desemnează o variabilă; evaluarea expresiei înlocuirea valorii variabilei indicate în membrul stîng cu valoarea obținută în membrul drept.

Membrul stîng al unei instrucțiuni de asignare este obligatoriu o variabilă (deci, poate fi și o înregistrare sau un element al unei înregistrări, un tablou sau un element al unui tablou, o mulțime etc.). De exemplu asignările:

INTEGER:= *5*
5:= *x*
x+*y*+*z*:= *20*

sînt incorecte, deoarece, în primul caz, în membrul stîng este numele unui tip de date, iar în al doilea, în membrul stîng se află o constantă; în al treilea caz în membrul stîng se găsește o expresie.

Următoarele asignări :

x:= *x* +*y***MOD***Z* ;
H:= *oră*+*minut*+*secundă* ;

sînt corecte dacă pentru fiecare asignare termenii din membrul drept definesc expresii corecte.

Pentru asignarea expresiilor este necesar să se țină seama de următoarele reguli: între date de tipuri diferite nu se realizează operații de asignare; în formarea expresiilor, operațiile de comparare între elementele unui tip produc valori boolene.

2.6. Structuri de control

Programarea structurată din MODULA-2 definește structuri de control formate din: execuții condiționate de instrucțiuni; execuții repetitive de instrucțiuni; grupări de instrucțiuni și date în proceduri.

MODULA-2 are următoarele instrucțiuni:

Instrucțiuni:= *asignare* | *apelproceduri* | **IF** | **CASE** | **WHILE** | **REPEAT** | **LOOP** | **FOR** | **EXIT** | **RETURN** | **WITH**.

O structură de control este realizată dintr-o secvență de instrucțiuni. O astfel de structură se execută secvențial, instrucțiune după instrucțiune, după ce precedenta s-a efectuat în ordine logică.

Dacă o instrucțiune I2 urmează după instrucțiunea I1 atunci această secvențialitate va fi indicată prin semnul de punctuație ";" (punct și virgulă), scriindu-se I1 ;I2. Acest semn de punctuație este separator de instrucțiuni și nu un simbol terminator. El indică faptul că acțiunea

specificată de I1 este urmată imediat de acțiunea corespunzătoare lui I2. Deoarece instrucțiunea de asignare a fost descrisă anterior le vom detalia pe celelalte.

2.6.1. *Instrucțiunea IF*

Permite programatorului să specifice secvențe de instrucțiuni executate alternativ în funcție de valoarea TRUE/FALSE a unei expresii de control.

Sintaxa este:

InstrucțiuneIF::= **IF** *Expresie* **THEN** *SecvențăInstrucțiuni1*
[**ELSIF** *Expresie* **THEN** *SecvențăInstrucțiuni2*]
[**ELSE** *SecvențăInstrucțiuni3*] **END** ";"

unde *Expresie* are o valoare booleană;

SecvențăInstrucțiuni::= *Instrucțiune* {";" *Instrucțiune*}

Exemplu:

```
MODULE Exif;
BEGIN
IF x=z; THEN
   x:=0;
ELSIF x=y; THEN x:=1;
ELSE
    y:=y-1
END;
END Exif.
```

2.6.2. *Instrucțiunea CASE*

Deși nu atît de generală ca instrucțiunea IF, ea este preferată în cîteva situații datorită expresivității. Al doilea motiv de utilizare se referă la eficiență, compilatorul producînd un cod mai bun. Sintaxa este:

InstrucțiuneCASE::= **CASE** *Expresie* **OF** {*CazuriPosibile* "|"}
[**ELSE** *SecvențăInstrucțiuni*] **END** ";"
CazuriPosibile::= *ListăEticheteCase*:*SecvențăInstrucțiuni*
ListăEticheteCase::= *Constantă*[: . *Constantă*]

Într-o instrucțiune CASE există o singură *Expresie*. Valoarea expresiei este comparată alternativ cu constantele sau intervalele de enumerare din lista de etichete și în caz de egalitate se execută secvențele de instrucțiuni corespunzătoare. *Exemplu:*

Din modulul *„Extern"* se importă constanta întreagă *x*. Cu operatorul **„MOD"** aflăm restul împărțirii la 4 a lui *x* și pentru fiecare caz în parte se notifică rezultatul cu procedura *WriteString*.

Scanned with OKEN Scanner

```
MODULE ExCASE;
FROM Extern IMPORT x;
FROM InOut IMPORT WriteString;
VAR
    X,i:CARDINAL;
BEGIN
        i:= x.Extern MOD 4;
        CASE i OF
              1:WriteString ('x.Extern dă rest 1 la împ. cu 4')
              2:WriteString ('x.Extern dă rest 2 la împ. cu 4')
              3:WriteString ('x.Extern dă rest 3 la împ cu 4')
        ELSE WriteString ('x.Extern divizibil cu4');
        END (*CASE*)
END ExCASE.
```

Se observă folosirea indicatorului de componenta selectată *x. Extern* pentru evitarea unor posibile confuzii cu alte constante, care ar putea fi definite în cadrul modulului.

Facem observația că în *ListăEticheteCase* nu este permisă apariția a două constante egale iar valorile luate de *Expresie* trebuie să fie de un tip compatibil cu tipul constantelor din *ListăEticheteCase*. De asemenea nu este permis ca aceste valori să fie de tipul REAL.

2.6.3. *Instrucțiunea WHILE*

Provoacă execuția repetată a unei secvențe de instrucțiuni atît timp cît o anumită condiție este adevărată.

Sintaxa este:

InstrucțiuneWHILE::=**WHILE** *Expresie* **DO** *SecvențăInstrucțiuni* **END**, ";"

Expresie trebuie să producă o anumită valoare booleană.

Dacă *Expresie* este adevărată, atunci se execută *SecvențăInstrucțiuni* și procesul se repetă, iar dacă este falsă, atunci controlul trece la prima instrucțiune după END.

Un grup de instrucțiuni ce se execută repetat formează un ciclu. O trecere a unui ciclu poartă numele de iterație. Folosirea instrucțiunii WHILE este indicată atunci cînd numărul de iterații ale ciclului nu este cunoscut în prealabil. În situația că numărul de iterații se cunoaște, este preferabilă folosirea instrucțiunii FOR.

Exemplu de modul-program prin care se calculează cîtul q și restul r al împărțirii a două numere întregi, x și y:

```
MODULE CIT;
    FROM InOut IMPORT WriteCard, WriteLn;
    VAR
    x,y,q,r: INTEGER;
    q:= 0;
```


Scanned with OKEN Scanner

```
        WHILE r <= x DO
            r:= q*y;
            q:= q+1;
        END;
            r:= x-q*y;
            WriteCard (q,10); WriteLn; WriteCard (r,10);
END CIT.
```

2.6.4. *Instrucțiunea REPEAT*

Este similară instrucțiunii WHILE cu cîteva diferențe:

— condiția pentru ciclare la instrucțiunea WHILE este la început, pe cînd la REPEAT condiția ciclului se află la sfîrșit;

— într-un ciclu WHILE iterația continuă atît timp cît condiția este adevărată, pe cînd într-un ciclu REPEAT iterația continuă cît timp condiția este falsă.

Sintaxa este:

InstrucțiuneREPEAT::= **REPEAT** *SecvențăInstrucțiuni* **UNTIL** *Expresie* ";"

Expresie are o valoare booleană.

Exemplu:

Se presupune că se citește un șir de caractere, între care s-a introdus un număr de caractere blank, din care 10 consecutive. Vom scrie un program ce va determina după cîte caractere se va găsi acest grup de zece caractere blank:

```
MODULE CaractereBlank;
    FROM Error IMPORT CitCard, Flag;
        (*CitCard este o procedură de citire a unui caracter*)
        (*Flag este o variabilă Booleană care are valoarea*)
        (*TRUE cînd se citește un caracter blank*)
        (*FALSE cînd se citește un caracter#blank*)
    FROM InOut IMPORT WriteCard, WriteLn;
    VAR
        Contor, NCblank: CARDINAL;
        (*Contor este număr caractere citite din care NCblank nr. caractere blank*)
    BEGIN
        Contor:=0;
        NCblank:=0;
        REPEAT
            Contor:= Contor+1;
            CitCard;
            IF Flag THEN
                NCblank:= NCblank+1;
            ELSE
                NCblank:= 0;
            END;(*IF*);
        UNTIL NCblank:= 10;
    WriteCard (Contor,4); WriteLn;
END CaractereBlank.
```


Scanned with OKEN Scanner

2.6.5. *Instrucțiunea FOR*

Este de fapt un ciclu specializat în care o variabilă de control ia valori succesive cuprinse între limite date.

Sintaxa este:

InstrucțiuneFOR::=
FOR *Identificator*:= *Expresie1* **TO** *Expresie2* **BY** *ExpresieConstantă*
DO *SecvențăInstrucțiuni* **END** "; "

Identificator este un identificator denumit în mod obișnuit variabilă de control. Valorile lui *Expresie1* și ale lui *Expresie2* sînt limitele variabilei de control și trebuie să fie compatibile cu tipul acesteia. *ExpresieConstantă* este un increment sau decrement adăugat variabilei de control la sfîrșitul fiecărei iterații din ciclu. Dacă **BY** *ExpresieConstantă* este omisă se presupune o incrementare cu 1. *SecvențăInstrucțiuni* se execută într-un cicluFOR o dată pentru fiecare valoare a variabilei de control.

Exemplu:

Se consideră modulul-program de însumare a pătratelor numerelor naturale de la 1 la 100:

```
MODULE Sumăsută;
    FROM InOut IMPORT WriteCard, WriteLn;
        VAR
            Sum, i: CARDINAL;
BEGIN
            Sum:= 0;
        FOR i:= 1 TO 100 DO
            Sum:= Sum+i*i;
        END; (*sfîrșit de ciclu FOR*)
    WriteCard (Sum, 5); WriteLn;
END Sumăsută.
```

La utilizarea ciclului FOR este necesară respectarea următoarelor reguli:

- limitele expresiilor nu trebuie să depindă de alte elemente din ciclu;
- este interzisă modificarea valorii variabilei de control în corpul ciclului.
- valoarea variabilei de control trebuie considerată nedefinită după terminarea ciclului.
- Variabila de control nu poate fi o variabilă importată, un parametru al unei proceduri, o înregistrare, un tablou sau o mulțime.
- variabila de control poate fi de tip CARDINAL, INTEGER, BOOLEAN, enumerativ, subdomeniu.

Scanned with OKEN Scanner

2.6.6. *Instrucțiunile LOOP și EXIT*

Ciclurile realizate cu instrucțiunile REPEAT, WHILE și FOR sînt similare cu structurile de control din PASCAL.

Instrucțiunea LOOP permite programatorului un control foarte bun asupra ciclurilor din program, fiind cea mai generală instrucțiune iterativă.

Sintaxa este:

InstrucțiuneLOOP::= **LOOP** *SecvențăInstrucțiuni* **END** ";"

Sfîrșitul unui ciclu LOOP este provocat de o instrucțiune EXIT inserată în *SecvențaInstrucțiuni*.

Absența instrucțiunii EXIT crează un ciclu infinit. Ciclurile infinite sînt deosebit de utile în cazul programării în timp real. De asemenea în *SecvențaInstrucțiuni* instrucțiunea EXIT poate să apară de mai multe ori, ceea ce creează condiții multiple de terminare a ciclului.

Cu instrucțiunea LOOP se pot rescrie toate instrucțiunile REPEAT, WHILE și FOR. De exemplu, programul dat la instrucțiunea WHILE se poate rescrie astfel:

```
MODULE CIT;
FROM InOut IMPORT WriteCard, WriteLn;
VAR
x,y,q,r:INTEGER;
BEGIN
q:= 0;
LOOP
r:= q*y;
q:= q+1;
IF r<=x THEN r:= x-q; EXIT END
END; (*LOOP*)
WriteCard(q, 10); WriteLn;WriteCard (r,10);
END CIT.
```

Instrucțiunile WITH și RETURN vor fi descrise în capitolele ce tratează înregistrări și, respectiv, proceduri.

CAPITOLUL 3

TIPURI DE DATE PREDEFINITE

Prin tip de date în limbajul MODULA-2 ca și în alte limbaje se înțelege o mulțime de elemente sau obiecte, denumită domeniu asupra căruia acționează anumite operații. Orice variabilă, constantă sau valoare de funcție din cadrul programului trebuie să aparțină în mod unic

Scanned with OKEN Scanner

unui anumit tip de date, apartenență care se deduce din modul de declarare a variabilelor, constantelor sau valorilor respective. Reciproc, un anumit tip de date definește toate proprietățile variabilelor, constantelor și valorilor pe care le conține. Constantele se presupun a fi valorile pe care le pot lua variabilele.

3.1. Clasificarea tipurilor de date

O clasificare a tipurilor de date recunoscute de MODULA-2 se poate face în ordinea următoare:

— Tipuri de date predefinite — pentru care nu mai este necesară declararea lor, fiind considerate declarate la implementarea limbajului. Acestea sînt:

- CARDINAL, LONGCARD, INTEGER, LONGINT, REAL, LONGREAL, BOOLEAN numite și tipuri elementare;
- BITSET, PROC, denumite și tipuri predefinite de tip mulțime și, respectiv, procedural;

— tipuri de date statice — create la compilarea programului, valabile pe întreaga durată a programului:

- nestructurate (scalare), ale căror domenii sînt constituite din obiecte independente. Aceste tipuri sînt cele enumerative și, respectiv, subdomeniu;
- structurate, ale căror domenii sînt constituite din entități care ele însele sînt obținute din obiecte ale altor tipuri. Aceste tipuri sînt mulțimi, tablouri, înregistrări;

— tipuri de date dinamice — create sau distruse la execuția programului. Domeniul lor conține obiecte care pot referi sau accesa informații ale altor obiecte. Din acest motiv se mai numesc și tipuri de acces.

— tipuri de date procedurale — ale căror domeniu și operații sînt definite de programator printr-o secvență program.

3.2. Declararea tipurilor

În MODULA-2 o declarație care specifică un tip definit de utilizator precizează mai întîi numele (identificator) pe care îl va purta, precum și construcția lui. Tipurile structurate sînt construite prin compunerea tipurilor recunoscute de limbaj după tehnici care vor fi explicitate pe parcursul lucrării. Sintaxa declarării unui tip este:

DeclarațieTip::= **TYPE** *identificator* = *Tip* " ; "

Tip::= *TipScalar* | *TipMulțime* | *TipTablou* | *TipÎnregistrare* | | *TipAcces* | *TipProcedural*.

Utilitatea practică a conceptului de tip este evidențiată prin faptul că se poate face o împărțire a variabilelor și constantelor programului într-o mulțime de clase disjuncte. Astfel, cresc facilitățile de întreținere și verificare, fără a exista posibilitatea unor asignări sau operații între elementele unor tipuri diferite, chiar dacă acestea sînt compatibile. Două tipuri de date sînt compatibile dacă sînt echivalente (declarațiile lor diferă numai prin identificator de nume) sau satisfac reguli de compatibilitate (admit aceleași operații precum și domenii de aceeași natură).

3.3. Tipuri elementare

Aceste tipuri sînt predefinite în cadrul limbajului, din care motiv nu mai este necesară declararea lor. Deși sînt similare limbajului PASCAL, ele vor fi detaliate în continuare deoarece sînt utilizate în toate programele.

3.3.1. *Tipul INTEGER*

Domeniul acestui tip este format din numere întregi care pot avea valori pozitive sau negative. O variabilă corespunzătoare tipului INTEGER desemnează numere întregi cuprinse între **MININTEGER** și **MAXINTEGER**. La minicalculatoare, **MININTEGER** = —32 768 și **MAXINTEGER** = 32 767.

Operatorii acestui tip, +, —, *, **DIV**, **MOD**, au următoarea semnificație: + este operația de adunare a numerelor întregi, — este operația de scădere a numerelor întregi, * este operația de înmulțire a numerelor întregi, **DIV** — operația de aflarea cîtului a două numere întregi, iar **MOD** — operația de aflare a restului împărțirii a două numere întregi

Exemplu: *7* **DIV** *3* are valoarea *2*, iar 9 **MOD** *5*, valoarea *4*.

Aceste operații pot produce valori în afara domeniului; ca urmare, la compilare sau la execuție se semnalizează depășirea.

În formarea expresiilor, operațiile menționate — numite și operații aritmetice — nu se combină cu rezultatul operațiilor de comparare care dau valori booleene, deci de alt tip. De exemplu, (3<7) **DIV** (5<9) este o expresie incorect formată, deoarece 3<7 și 5<9 sînt expresii de tip boolean, iar operația **DIV** între două valori booleene nu are sens.

Există și cîteva funcții puse la dispoziție de compilatorul MODULA-2 care au argumente variabile de tip întreg (INTEGER); de pildă, funcția **ABS** ce furnizează valoarea absolută sau funcția **ODD** ce indică paritatea sau imparitatea unui număr.

Scanned with OKEN Scanner

Exemplu:
ABS (*—300*)=*300*;
ODD (*25*) = **TRUE**;

De asemenea, menționăm funcțiile **INC** și **DEC** folosite pentru incrementarea, respectiv, decrementarea unei variabile sau constante de tip INTEGER:

INC (*temperatura*, 4); crește *temperatura* cu *4*
DEC (*temperatura*, 2); scade *temperatura* cu *2*

3.3.2. *Tipul CARDINAL*

Domeniul acestui tip este format din numere naturale, iar operațiile sînt aceleași ca la tipul INTEGER. Trebuie menționat că funcția **ABS** nu este permisă, iar la minicalculatoare **MAXCARDINAL** = 65 535 și **MINCARDINAL** = 0. Se pot folosi expresii ce conțin simultan tipuri **CARDINAL** și **INTEGER** numai prin utilizarea funcțiilor de conversie între tipuri, după cum se arată în tabelul II.1.

3.3.3. *Tipul REAL*

Domeniul acestui tip este format din numere cu parte fracționară sau zecimală. Reprezentarea în codul mașină nu corespunde cu valoarea exactă a numărului real, deoarece tehnicile de reprezentare internă au caracter limitativ. Toate operațiile de la tipul INTEGER sînt valabile și pentru tipul REAL. Spre deosebire de limbajul PASCAL nu este permisă aplicarea operațiilor de adunare și scădere între elementele tipului INTEGER și REAL. (Limbajul PASCAL are o tipizare mai slabă decît MODULA-2). Se poate utiliza funcția ABS pentru tipul REAL, dar nu pot fi folosite funcțiile ODD, INC și DEC.

Dacă se cere o mai mare acuratețe a valorilor manipulate se folosesc tipurile extinse LONGINT pentru numere întregi (la minicalculatoare **MAXLONGINT** = 2 147 483 648) și, similar, LONGCARD și LONGREAL pentru numere naturale, respectiv, reale.

3.3.4. *Tipul BOOLEAN*

Domeniul acestui tip este format din elementele TRUE și FALSE asupra cărora acționează operatorii logici AND, OR, NOT, &, ~. Operatorul „~" (tilda) este sinonim cu NOT, iar operatorul „&", cu AND. Operațiile sînt definite astfel:

*p***AND***q* = **IF***p***THEN***q***ELSEFALSEEND**;
*p***OR***q* = **IF***p***THENTRUEELSE***q***END**;
NOT*p* = **IF***p***THENFALSEEND**;

Scanned with OKEN Scanner

Acest tip se utilizează în scopul stabilirii valorilor de adevăr ale expresiilor și formalizării regulilor de decizie.

Valorile de tip **BOOLEAN** pot proveni din expresii ce conțin comparări de valori aparținînd altor tipuri (de exemplu, INTEGER, CARDINAL, CHAR, REAL și altele) și operații logice cu valori de tip **BOOLEAN**.

MODULA-2 recunoaște următorii operatori de comparare: =, #, <>, <, <=, >, >=, IN.
Operatorul „<>", similar cu „#", înseamnă inegalitate. Operatorul IN semnifică apartenența unui element la un obiect de tip SET (mulțime) (vezi capitolul 4).

Conform definiției, operațiile mai sus menționate trebuie să se conformeze următoarelor reguli:

- operanzii trebuie să fie de același tip;
- rezultatul unei comparări este obligatoriu o valoare booleană.

Exemple de expresii booleene:

$x = y;$
$(x <= y)\ \&\ (x = y);$
NOT p **AND** $q;$

Expresiile booleene se evaluează similar cu expresiile aritmetice, ținîndu-se seama de nivelul de precedență al operațiilor. Operatorul **NOT** are prioritate maximă, apoi urmează **AND** și, în continuare, **OR**. Parantezele se utilizează pentru explicitarea asocierii dintre operanzi și operatori.

Atragem de asemenea atenția asupra regulii conform căreia pentru orice operator operanzii trebuie să fie de același tip. De exemplu, 0=FALSE + 1; 5.0 + 5 = pORq sînt expresii incorecte, întrucît utilizează operații cu date aparținînd unor tipuri diferite.

3.3.5. *Tipul CHAR*

Domeniul acestui tip este format din caractere ASCII. Există unele implementări ale limbajului cu setul EBCDIC. Constantele acestui tip sînt caractere delimitate de (') sau de ("), de ex.: 'b', "B", 'X', "~". Este recunoscută și scrierea codurilor ASCII a elementelor (caractere urmate de sufixul C) acestui tip (ex.: 142C, 102C, 130C, 176C) utilizată mai ales pentru caracterele netipăribile (ex.: 40C pentru spațiu). În acest tip sînt implementate doar operațiile de asignare și comparare. Dacă **VAR** *ch*: **CHAR**; are sens expresia: (ch>='a') **AND** (ch<='z'); sau asignarea ch:="B", dar nu au sens următoarele construcții: "b"—"B"; sau 142C—102C.

Pentru folosirea codificării caracterelor ASCII, limbajul MODULA-2 conține funcția **ORD** ( ) care asociază unui caracter codul. Reciproc, oricărui număr $0 <= n < 177$ funcția **CHR** ( ) îi asociază un caracter ASCII. Aceste funcții complementare sînt legate prin relațiile:

CHR (**ORD**(*Car*)) = *Car* și **ORD** (**CHR**(*n*)) = *n*.

Ele permit calculul valorii numerice octale a unui caracter, „*car*", astfel:

ORD (*Car*) — **ORD** („*O*").

Reciproc, caracterul ASCII a cărui valoare octală este *n* va fi dat de expresia:

CHR (*n* + **ORD**(„*O*")).

O parte din caracterele tipului CHAR sînt utilizate drept caractere de control. Acestea sînt folosite pentru controlul funcțiilor dispozitivelor periferice, delimitarea și ordonarea structurii unui text. De exemplu, 15C este caracterul de control pentru întoarcerea carului (CR).

3.3.6. *Compatibilitatea tipurilor și funcții de conversie și transfer între tipuri*

Pentru asigurarea unei folosiri neambigue a tipurilor de date, în MODULA-2 s-au implementat mecanisme de **verificare strictă a tipurilor.** Aceasta înseamnă că tipurile sînt verificate în următoarele situații:

— la asignarea unei valori la o variabilă;
— la formarea expresiilor;
— la utilizarea elementelor aparținînd unui anumit tip ca argumente în proceduri;
— în sintaxa instrucțiunilor (în instrucțiunea FOR nu se admit expresii de tip real);
— la utilizarea indicilor în tipuri ARRAY (tablou).

Vom detalia acest mecanism la utilizarea diferitelor tipuri în aceeași expresie. Pentru formarea expresiilor programatorul trebuie să țină seama de următoarele reguli:

— compatibilitatea logică a expresiilor;
— compatibilitatea de asignare care se aplică în momentul asignării unei valori la o variabilă.

Compatibilitatea logică cere ca în cadrul aceleiași expresii să nu se folosească tipuri diferite decît după ce s-a asigurat conversia sau transferul între tipuri.

Scanned with OKEN Scanner

Compatibilitatea de asignare este mai puțin severă, în sensul că se pot face asignări între anumite tipuri diferite. De exemplu, tipurile CARDINAL și INTEGER sînt compatibile la asignare.

Vom exemplifica în următoarele cazuri:

Fie i și j variabile de tipul INTEGER și c și d variabile de tipul CARDINAL. Atunci următoarele expresii sînt corecte:

$$i := c+d;\ c:=i+j;$$

deoarece variabilele c și d sînt de tip CARDINAL, iar rezultatul este compatibil la asignare cu variabila i de tip INTEGER. Similar $i+j$ este compatibilă la asignare cu variabila c de tip CARDINAL.

Nu același lucru, se poate spune despre instrucțiunile:

$$i := i+c;$$
$$c := i+c;$$

care nu sînt corecte întrucît, expresia $i+c$ nu respectă regula de compatibilitate logică.

Pentru relaxarea condițiilor de incompatibilitate între tipuri, MODULA-2, conține funcții de conversie între tipuri, care permit ca elementele unui tip să fie convertite în elementele altui tip, atunci cînd este posibil. Aceste funcții de conversie au ca argument un element al unui tip și returnează ca valoare un element al altui tip conform tabelului:

Tabelul II.1

Funcții de conversie între tipurile predefinite din MODULA-2

Tipul argumentului	Funcția	Tipul rezultatului
a:= cardinal	FLOAT (a)	REAL
a:= real	TRUNC (a)	CARDINAL
a:= cardinal	CHR (a)	CHAR
a:= caracter	ORD (a)	CARDINAL
a:= întreg	ORD (a)	CARDINAL

În afara funcțiilor de conversie există posibilitatea interpretării conținutului locației de memorie asociat elementului unui anumit tip ca fiind valoarea unui element aparținînd altui tip de date. Acest procedeu se numește transferul valorii între elementele a două tipuri și limbajul pune la dispoziție funcții de transfer.

Scanned with OKEN Scanner

Pentru toate tipurile între care se pot face transferuri există posibilitatea de a stabili regula generală conform căreia numele tipului de destinație poate constitui numele funcției de transfer.

Exemplu:

VAR *Ival*: **INTEGER**; *Cval*: **CARDINAL**;

Putem forma atunci următoarea expresie:

INTEGER (*Cval* + **CARDINAL**(*Ival*));
Ival:= **INTEGER** (*Cval*);

3.4. Tipul BITSET

Domeniul acestui tip este format din șiruri de numere naturale. Operațiile corespunzătoare sînt: + reuniune de șiruri; — diferență de șiruri; * intersecție de șiruri; / diferență simetrică; IN incluziune de șiruri.

Șirurile de acest tip se scriu ca o secvență neordonată de numere naturale cuprinse între acolade de exemplu, {*1, 2, 7, 9, 12*}, {*5, 25, 17, 19, 27*}. Operatorul IN trebuie văzut ca un operator extern, în sensul că are argumente numere naturale și elemente din **BITSET**, iar valorile sînt de tip BOOLEAN.

Dacă se presupune că *i* reprezintă un număr natural, iar *u* și *v* șiruri din BITSET, atunci *i* **IN***u* are valoarea **TRUE** cînd *i* este inclus în *u*, iar cînd *i* nu este inclus în *u* are valoarea **FALSE**.

Operațiile tipului BITSET sînt definite astfel:

- *i***IN**(*u*+*v*)=(*i***IN***u*) **OR** (*i***IN***v*), adică *i* se află în reuniunea *u*+*v* dacă este sau în *u* sau în *v* sau în amîndouă;
- *i***IN**(*u*—*v*)=(*i***IN***u*) **AND NOT** (*i***IN***v*), adică *i* este în diferența *u*—*v* dacă se află în *u*, dar nu în *v*;
- *i***IN**(*u***v*)=(*i***IN***u*) **AND** (*i***IN***v*), adică *i* este în intersecția *u***v* dacă se află atît în *u*, cît și în *v*;
- *i***IN**(*u*/*v*)=((*i***IN***u*)<>(*i***IN***v*)), adică *i* se găsește în diferența simetrică *u*/*v* dacă este sau în ambele șiruri *u* și *v* sau în niciunul.

Acest tip se utilizează pentru identificarea pozițională a biților din cadrul unui cuvînt din memoria calculatorului. De exemplu, al *n*-lea bit din *u* este 1 dacă *n***IN***u*=**TRUE**; dacă *n***IN***u*=**FALSE**, bitul este 0. Cuvîntul *u* nu este altceva decît șirul format din biții {0, 1, 2, ..., *n*}.

Scanned with OKEN Scanner

CAPITOLUL 4

TIPURI DE DATE STRUCTURATE

În afară de tipurile predefinite descrise în capitolul 3, MODULA-2 pune la dispoziția programatorului mecanisme de construcție, pornind de la acestea, a noi tipuri destinate să organizeze colecții de date necesare prelucrării algoritmice. În acest sens pot fi construite tipuri enumerative (liste de elemente ale domeniilor unor tipuri) și tipuri subdomeniu (formate din secvențe de elemente ordonate ale domeniilor unor tipuri).

Dacă în plus grupăm elemente din domeniul unui tip pentru a forma noi elemente de sine stătătoare se obțin obiecte structurate.

În MODULA-2 există trei tipuri ale căror domenii sînt constituite din obiecte structurate. Aceste tipuri pot fi clasificate astfel:

— tipuri ARRAY (tablouri), definite de grupe de elemente de același tip indexate;

— tipuri RECORD (înregistrări), definite de grupe de elemente de tipuri diferite;

— tipuri SET (mulțimi), definite ca familii de mulțimi formate din elementele unui anumit tip.

4.1. Tipuri de date enumerative (liste elementare)

Domeniul unui tip enumerativ este constituit dintr-o listă de elemente date prin valorile lor (numerice sau literale).

Operațiile admise sînt de asignare și comparare.

Declararea unei liste elementare se face simplu prin denumirea elementelor ce le conține, de exemplu:

```
TYPE
    CULORI = (Roșu, Verde, Albastru, GALBEN, Negru, Maro, gri);
```

În consecință declararea tipului enumerativ se face prin enumerarea valorilor pe care le pot lua elementele acestui tip.

DeclarațieTipEnumerativ: : =

TYPE *numelistă* = "("{*valoareelement*}")";

După definirea listelor se pot face asignări între elemente diferite ale aceleiași liste, dar sînt interzise asignările elementelor din liste cu elemente ale altei liste sau ale altui tip.

Scanned with OKEN Scanner

De exemplu, asignările:

Roșu := *culoare*; sau
Maro := *25*;

sînt incorecte atît timp cît *culoare* nu e definită în nici o listă, iar 25 este de tip INTEGER.

Conform definiției sînt permise operații de comparare între elementele unei liste, dar și alte operații externe puse la dispoziție o dată cu implementarea limbajului sau de către utilizator. De exemplu, procedura **INC** atribuie unei variabile o valoare următoare din listă, procedura **DEC** atribuie unei variabile o valoare anterioară din listă, iar procedura **ORD** determină poziția unei valori dintr-o listă.

Exemple:

VAR x; *CULORI*;
x: = *Roșu*
INC (*x*, 3) <=> *x*: = *Albastru*;
DEC (*x*, 3) <=> *x*: = *Roșu*;
ORD (*Roșu*) = 0, **ORD** (*GALBEN*) = 3, din lista *CULORI*.

Pentru a putea atribui unei variabile o valoare dintr-o listă a cărei poziție o cunoaștem se poate folosi procedura **VAL** după cum rezultă din următorul exemplu:

Dacă în cadrul tipului *CULORI* se declară următoarele variabile:

VAR
pepene, ardei, cirese, cer:*CULORI*;

se pot efectua următoarele asignări echivalente:

pepene := *GALBEN*; sau
pepene := **VAL** (*CULORI*,4);

Tipul elementar BOOLEAN este și el un tip enumerativ ce a fost declarat ca o listă cu două valori FALSE și TRUE.

TYPE BOOLEAN = (FALSE, TRUE);

iar operațiile interne sînt cele ale algebrei logice. De asemenea, tipurile elementare predefinite, mai puțin REAL și LONGREAL din cadrul limbajului, sînt și ele tipuri enumerative.

4.2. **Tipuri subdomeniu** (Liste secvențiale)

Pentru a înlesni manipularea unor liste este util să se folosească șiruri de elemente succesive ale unui tip enumerativ, provenit din tipurile elementare: INTEGER, CHAR, CARDINAL. De exemplu, ne interesează să folosim toate valorile întregi cuprinse în intervalul [—25,

Scanned with OKEN Scanner

+25]. Putem declara subdomeniul definit de acest interval și construi un tip de date prin declarația:

```
TYPE
    D = [—25 .. 25];
```

. Vom utiliza variabile ale acestui subdomeniu cu declarația:

```
VAR
    a, b, c, d: D;
```

Trebuie reținut că datele din subdomeniu sînt formate din toate elementele tipului din care provin, cuprinse între cele două valori ce limitează subdomeniul. Tipul subdomeniu este compatibil (admite aceleași operații) cu tipul din al cărui domeniu este construit. În cazul anterior D este compatibil cu tipul INTEGER.

Sintaxa declarării tipului subdomeniu este:

DeclarațieTipSubdomeniu:=**TYPE**
NumeSubdomeniu = "["*ExpresieConstantă1*. .*ExpresieConstantă2*"]" ";"

ExpresieConstantă1 și *ExpresieConstantă2* sînt limitele subdomeniului, fiind constante ce aparțin unuia din tipurile INTEGER, CARDINAL, CHAR și enumerativ, iar *NumeTipSubdomeniu* este identificatorul de nume al subdomeniului.

Exemple:

```
TYPE
    L = ["A" . . "P"]
```

Tipul *L* conține toate literele din alfabet cuprinse între *A* și *P*. Se observă deci că cele două limite aparțin tipului CHAR;

```
TYPE
    ZILE = (Luni, Marți, Miercuri, Joi, Vineri, Sîmbătă, Duminică); (*tip
           enumerativ*)
```

Putem declara tipul subdomeniu ZILELUCRĂTOARE astfel:

```
TYPE
    ZILELUCRĂTOARE = [Luni . . Sîmbătă]; (*tip subdomeniu*)
```

4.3. Tipul SET (Mulțime)

Din punct de vedere matematic o mulțime este o colecție de obiecte. În MODULA-2 o mulțime este un tip de date ale cărui obiecte sînt constituite din grupe de elemente ale unui tip enumerativ sau subdomeniu, iar operațiile interne sînt reuniunea, diferența, intersecția și diferența simetrică.

Scanned with OKEN Scanner

Sintaxa declarării tipului mulțime este:

DeclarațieTipMulțime::=

TYPE *NumeMulțime* = **SET OF** *TipListă*";"

TipListă este un tip enumerativ sau un tip subdomeniu care trebuie anterior declarat; el constituie tipul „suport" al tipului mulțime.

În exemplul următor declarăm tipurile mulțime1, mulțime2 conform sintaxei de mai sus:

```
MODULE Mulțimi;
TYPE (*tip enumerativ*)
     Listă = (Roșu, Verde, Albastru, Galben, Violet, Ocru);
TYPE
     Sublistă = [Albastru.. Ocru]; (*tip subdomeniu*)
     mulțime1 = SET OF Listă;   (*tip mulțime*)
     mulțime2 = SET OF Sublistă; (*tip mulțime*)
CONST
     C1 = Listă {Roșu};
     C2== Listă {Roșu.. Albastru};
     C3 = Listă {Albastru, Violet};
     C4 = Sublistă {Albastru, Galben};
BEGIN
END Mulțimi.
```

Acest exemplu arată construcția sintactică a indicatorilor de constante ale unei mulțimi, pornind de la elementele tipului „suport" al acestei mulțimi (vezi anexa II.1).

În *mulțime1* sînt declarate constantele *C1*, *C2*, *C3*, iar în *mulțime2* este declarată constanta *C4*.

C1 conține un element, *C2* trei elemente, *C3* două elemente iar *C4* două elemente. Se observă că pentru *C2* s-a folosit notația enumerativă.

Tipul BITSET întîlnit în capitolul 3 ar putea fi definit astfel:

TYPE BITSET = SET OF [*0..n—1*];

unde *n* este lungimea cuvîntului calculatorului pe care este implementat compilatorul MODULA-2.

Pe calculatoarele cu lungimea cuvîntului de 16 biți (INDEPENDENT, CORAL) tipul BITSET este definit astfel:

TYPE BITSET = SET OF [*0..15*];.

BITSET este foarte important în programarea de sistem, deoarece poate fi utilizat pentru accesul la locații fizice de memorie.

Operații cu mulțimi

Fiind date două mulțimi *A* și *B* de același tip, cele patru operații permise sînt date de următorii operatori:

Scanned with OKEN Scanner

- operatorul + pentru operația de reuniune;
- operatorul − pentru operația de diferență;
- operatorul * pentru operația de intersecție;
- operatorul / pentru operația de diferență simetrică.

Sensul logic al acestor operatori poate fi exploatat în MODULA-2 astfel:

x **IN** (*A*+*B*) = (*x* **IN** *A*) **OR** (*x* **IN** *B*)
x **IN** (*A*−*B*) = (*x* **IN** *A*) **AND** (**NOT** (*x* **IN** *B*))
x **IN** (*A***B*) = (*x* **IN** *A*) **AND** (*x* **IN** *B*)
x **IN** (*A*/*B*) = (*x* **IN** *A*) <> (*x* **IN** *B*)

Pentru mulțimi operatorul **IN** certifică sau nu apartenența unui element la o mulțime.

De asemenea, sînt puși la dispoziția programatorului următorii operatori binari externi definiți pe un tip mulțime și cu valori în tipul BOOLEAN:

- operatorii = și <> sau # pot fi utilizați pentru testarea egalității sau inegalității celor două mulțimi;
- operatorii => sau <= pot fi utilizați pentru testarea incluziunii. De exemplu, *A* => *B* are valoarea TRUE cînd *A* include mulțimea *B*. Pentru testarea apartenenței unui element la o mulțime se utilizează operatorul IN. El cere o expresie aparținînd unui tip mulțime în partea dreaptă, iar în partea stîngă o expresie a tipului enumerativ din care provine expresia din partea dreaptă. De altfel, operatorul IN este singurul operator care cere doi operatori de tipuri diferite. De exemplu, 7 IN {6..12} are valoarea adevărată deoarece 7 este un element al mulțimii formate din numerele 6, 7, 8, 9, 10, 11, 12. Similar, procedurile INCL și EXCL reunesc elementele a două mulțimi respectiv elimină elementele unei mulțimi din altă mulțime.

Exemplu:

```
MODULE exinc;
  TYPE
       CULORI = (Roșu, Verde, Albastru);
       Listăelem = SET OF CULORI;
  VAR
       materiale: Listăelem;
       pălărie:CULORI;
       cămașă:CULORI;
  BEGIN
       pălărie:= Verde;
       materiale: = listăelem {Roșu, Albastru};
       EXCL (materiale, Roșu);
       INCL (materiale, pălărie);
       IF materiale: = listăelem {Verde, Albastru} THEN
          materiale: = listăelem {Verde} / listăelem {cămașă};
```


Scanned with OKEN Scanner

```
        ELSE
          materiale: = listăelem {pălărie} + materiale;
        END
    END exinc.
```

Modulul exinc conține tipul mulțime listăelem construit din elementele Roșu, Verde, Albastru ale tipului enumerativ CULORI.

4.4. Tipuri de date ARRAY (Tablouri)

Declararea unui tip ARRAY trebuie să menționeze cel puțin două tipuri, și anume tipul elementelor din tablouri și, respectiv, al indexului, cu precizarea limitelor indexului cînd contextul o cere.

Sintaxa declarării tipului este:

DeclarațieTipTablou::=

```
TYPE NumeTablou: = ARRAY tipindex OF tip;
```

unde *tip* este orice tip de date recunoscut de MODULA-2, inclusiv tablou;

tipindex este unul din următoarele tipuri de date: INTEGER, CARDINAL, CHAR, BOOLEAN, enumerativ și subdomeniu.

Domeniul unui tip „Tablou" este constituit din structuri de obiecte ale tipului de date *„tip"*. Aceste obiecte sînt indexate după valori ale tipului de date *„tipindex"*. Fiecare structură se numește tablou de tip *„Numetablou"*.

Exemplu:

```
TYPE Tablou0 = ARRAY [1. .90] OF CHAR;
TYPE Tablou1 = ARRAY [30. .200] OF CHAR;
TYPE Tablou2 = ARRAY [1. .IndexMax] OF REAL;
TYPE Tablou3 = ARRAY CARDINAL OF BOOLEAN;
```

Tablou0 este un tip de date tablou format din structuri cu cîte 90 de elemente de tip caracter. Tipul index din *Tablou1* este un tip subdomeniu al tipului INTEGER. La fel este definit și *Tablou2*, cu mențiunea că valoarea lui *IndexMax* trebuie precizată.

Definirea lui *Tablou3* conține întreg tipul CARDINAL drept tip index (bineînțeles că *Tablou3* nu va conține o infinitate de elemente deoarece capacitatea calculatorului este limitată). De fapt același lucru este valabil și pentru tipurile predefinite REAL, INTEGER, CARDINAL ce ar putea avea o infinitate de elemente și pe care calculatorul nu are suficiente resurse pentru a le memora. După cum s-a mai menționat un tip „Tablou" nu admite constante. Calificarea unui tablou se face atribuind valori fiecărui obiect al structurii din care este con-

Scanned with OKEN Scanner

stituit. Presupunem că dispunem de o variabilă de tip *Tablou1*, care a fost anterior declarat:

VAR *T*: *Tablou1*;

Un element din cadrul tabloului T se indică prin menționarea identificatorului de tablou „T" urmat de valoarea indexului cuprins între paranteze drepte.

Exemplu: *T [40] : = 60C*; În acest exemplu „T [40]" este indicatorul elementului „40" din tabloul T, căruia i se atribuie valoarea „60C".

În MODULA-2 sînt permise următoarele operații cu tablouri:

- asignarea unui tablou altui tablou de același tip;
- utilizarea tablourilor ca parametri în proceduri.

Subliniem că nu sînt permise următoarele operații:

- operația de comparare între tablouri;
- adunarea sau reuniunea a două tablouri;
- returnarea unui tablou ca valoare a unei proceduri funcționale.

Procedurile pot avea ca parametri tablouri de date.

Eexemplu:

```
MODULE tablou;
        CONST
                Valmax = 99;
        TYPE
                Tablou = ARRAY [0. .Valmax] OF REAL;
        PROCEDURE Dem (x:Tablou): REAL;
        VAR
                i: CARDINAL;
                Sum: REAL;
        BEGIN
                Sum: = 0;
                  FOR i: = 0 TO Valmax DO Sum:= Sum +x[i] ;END;
                  RETURN (Sum/FLOAT(Valmax + 1))
        END Dem;
BEGIN
END tablou;
```

În cazul în care valoarea maximă a indexului de tablou nu este precizată, pentru controlul valorilor indexului se poate utiliza procedura HIGH de determinare a numărului elementelor din tablou, valoarea minimă presupunîndu-se zero.

Dacă dorim ca elementele unui tablou să fie la rîndul lor tablouri (și așa mai departe) putem realiza acest deziderat numai prin construirea unui tip compus, numit tablou bidimensional (multidimensional).

Exemplu:

```
TYPE
Tabloudim = ARRAY [0. .9] OF ARRAY [0. .99] OF CHAR;
```


Scanned with OKEN Scanner

definește un tip tablou ale cărui obiecte sînt matrice de 9 linii și 99 coloane, iar elementele matricelor sînt caractere.

Un element al tipului *Tabloudim*, de exemplu, cel de pe linia 4 și coloana 25, poate fi calificat prin instrucțiunea de asignare:

```
VAR
x: Tabloudim;
x [4] [25] := "a";
```

x [4] [25] este indicatorul elementului calificat.

Uneori pentru simplificare putem să declarăm numai variabilele tipului ca în următorul exemplu:

```
VAR
      x: ARRAY[0..9, 0..99] OF CHAR;
```

Și calificarea elementului de pe linia 4 și coloana 25 se face similar:

x [4,25]: = 'a'

x [4,25] este indicatorul elementului calificat.

4.5. Tipuri de date RECORD (Înregistrări)

Utilizarea tipurilor de date RECORD este utilă atunci cînd se dorește gruparea unui șir de variabile sau constante care nu sînt de același tip într-o singură entitate, numită înregistrare.

Sintaxa declarării acestui tip este:

DeclarațieTipÎnregistrare:: =
TYPE *NumeÎnregistrare*:= **RECORD** *ListădeCîmpuri* **END**";"
ListădeCîmpuri:: ={*ListăIdentificatori*:*TipCîmp*";"}

ListăIdentificatori conține constante sau variabile aparținînd tipului de date *TipCîmp*; acesta poate fi orice tip recunoscut de MODULA-2.

Vom exemplifica două cazuri de structurare a datelor în înregistrări. Un tip de înregistrări conține caracteristicile unui costum bărbătesc — întreprinderea producătoare, preț, talie, culoare, fibră, model —, iar următorul luna și anul de fabricație. Acestea se exprimă prin următoarele declarații:

```
TYPE Costumhaineb =
        RECORD
                Intrprod : ARRAY [0..20] OF CHAR;
                preț, talie, model:CARDINAL;
                fibră, culoare:ARRAY [0..15] OF CHAR;
                dată: datăfabricație;
        END ;
TYPE datăfabricație =
        RECORD
                ziua : CARDINAL;
                luna: ARRAY [0..10] OF CHAR;
                anul: CARDINAL;
        END;
```

Referirea se poate face la o anumită înregistrare din cadrul tipului sau la un element individual din cadrul înregistrării.

Componentele unei înregistrări se numesc cîmpuri ale înregistrării, iar fiecare cîmp este referit printr-un indicator de cîmp.

Similar tablourilor (ARRAY), unde al n-lea element din tabloul A se notează cu indicatorul $A[n]$, adică identificatorul tabloului urmat de indexul elementului, indicatorul cîmpului C al înregistrării F este $F \cdot C$, adică identificatorul înregistrării urmat de "." și numele cîmpului.

Dacă pentru înregistrările *costumhaineb* și *datăfabricație* considerăm variabilele:

```
VAR C: Costumhaineb;
VAR D: datăfabricație;
```

cîmpurile corespunzătoare *Intrprod, dată, luna, anul* sînt exprimate cu următorii indicatori:

C.Intrprod
C.dată
D.luna
D.anul

Cîmpurile unei înregistrări pot fi și ele de tip ARRAY sau RECORD, caz în care, elementele cîmpului se indică astfel:

C.Intrprod[8]
C.dată. anul

În scopul calificării (atribuirea de valori componentelor) unei anumite înregistrări sau a unui cîmp dintr-o înregistrare, fără a se utiliza instrucțiuni de asignare a indicatorilor este pusă la dispoziția programatorului instrucțiunea WITH. Ea are următoarea sintaxă:

```
InstrucțiuneWITH::=
WITH VariabilădeControl DO SecvInstrucțiuni END;
```

VariabilădeControl este nume de înregistrare sau indicatorul unui cîmp dintr-o înregistrare ori a unui element dintr-un tablou de înregistrări.

SecvInstrucțiuni conține instrucțiuni care nu modifică variabila de control.

Vom califica o înregistrare a tipului *Costumhaineb* folosind instrucțiunea WITH:

```
MODULE Costumdate;
        TYPE
            Costumhaineb =
              RECORD
              Intrprod:ARRAY [0..20] OF CHAR;
              Preț, Talie, Model: CARDINAL;
              Fibră, Culoare: ARRAY [0..15] OF CHAR;
              dată:datăfabricație; END;
            datăfabricație =
```


Scanned with OKEN Scanner

```
            RECORD
            ziua: [1..31];
            luna: ARRAY [0..10] OF CHAR;
            anul: [1985..1990];
    END;
VAR
  costum406, costum305:Costumhaineb;
BEGIN
        WITH Costum305 DO
                Intrprod: = 'FCTB';
                Preț    : = 1200;
                Talie   : = 50;
                Model   : = 1987;
                Fibră   : = 'LINA 60% PAMD 40%';
                Culoare : = 'BLEUMARIN';
                data    : = 5 'MAI' 1987
        END;
                Costum406: = costum305;
                costum406.Model: = 1988;
                costum406. Fibră. 5  : =7;
                costum406. Fibră: 12  : =3;
                costum406. data: = 7 'IUNIE' 1987
END Costumdate.
```

Cîmpurile înregistrării al cărei identificator este *costum305* au fost calificate cu instrucțiunea WITH, pe cînd înregistrarea *costum406* s-a obținut din înregistrarea *costum305* în care am modificat cîmpurile Model, Fibră și data, folosind indicatori de cîmpuri. Putem obține același lucru prin folosirea instrucțiunii WITH. Trebuie amintit că este preferabilă calificarea cîmpurilor dintr-o înregistrare cu instrucțiunea WITH din motive de eficiență și lizibilitate.

Există posibilitatea de a califica un tablou de înregistrări, folosind ca variabilă de control a instrucțiunii WITH un element al tabloului de înregistrări.

Exemplu:

```
TYPE
      Costume1989 = ARRAY [0..2000] OF Costumhaineb;
VAR
      COSTUME : Costume1989;
FOR i : = 305 TO 408 DO
      WITH COSTUME [i] DO
            Model := 1988;
             Culoare := „bleumarin"
      END   (*WITH*)
END ; (*FOR*)
```

După cum se observă într-o buclă FOR putem folosi o instrucțiune WITH, dar reciproc nu este permis dacă modifică variabila de control. O utilizare care pune în evidență eficiența tipurilor RECORD este

Scanned with OKEN Scanner

generarea listelor cu legături multiple, structuri de date de mare importanță în planificarea resurselor. Listele cu legături multiple diferă de listele obișnuite prin faptul că numărul elementelor (înregistrări) este variabil și fiecare element (înregistrare) conține date ale altor elemente (înregistrări).

În exemplul următor se construiesc proceduri de manipulare liste cu legături multiple (fig. II.2.).

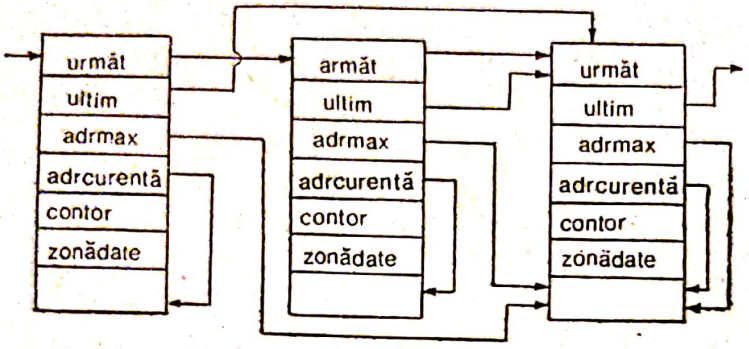


Fig. II.2. Exemplu de zone de date cu antet și legături multiple

Fiecare buffer conține un antet plus o zonă de date:

```
DEFINITION MODULE LLBuf;
(*modul de utilizare al listelor înlănțuite*)
TYPE
    LINK; (*Tip opac, index pentru înregistrări*)
PROCEDURE depozit (ch:CHAR; chan:LINK);
PROCEDURE extragere (VAR chan:LINK): : CHAR;
PROCEDURE contor (chan:LINK);
END LLBuf.
```

Prin apelul procedurii *Openbuf* se extrage din „pool" primul buffer al listei după care se actualizează antetul cu referințele buferului. Fiecare bufer este o înregistrare (tip RECORD), în timp ce „pool"-ul de buffere sau lista de înlănțuiri este un tip de date ARRAY OF RECORD, iar indexul buferelor din „pool", de tip enumerativ.

Procedura *depozit* adaugă un nou buffer în lista înlănțuită, pe cînd procedura *Extragere* elimină un buffer din lista înlănțuită, actualizînd indexul tipului enumerativ (LINK). Procedura *contor* modifică indexul de buffer. Parametrii procedurilor descriu elementele din antetul bufferelor, cum se poate vedea în fig. II.2. Indexarea se referă la adresa următorului buffer care, atunci cînd ia valoarea „0" indică sfîrșitul șirului de buffere. Indexul ultimei înregistrări este indexul ultimului buffer, iar *adrcurentă* este adresa următorului caracter accesibil; *adrmax* este indexul adresei curente din ultima înregistrare.

```
IMPLEMENTATION MODULE LLbuf;
FROM InOut IMPORT WriteString, WriteLn;
CONST
  SizeBuf = 31;
NrBufere = 32;
NULL = 0;
(*declararea completă a tipului enumerativ LINK*)
TYPE
    LINK = [0. .NrBufere];
(*declararea înregistrărilor din LanțBuf*)
```

```
TYPE
      LanțBuf =
          următ:LINK; (*capul și *)
          ultim:LINK; (*coada listei*)
          adrcurentă:CARDINAL; (*index de buffer*)
          adrmax:CARDINAL; (*index de buffer*)
          contor:CARDINAL; (*număr caractere din buffer*)
          data:ARRAY [0. .SizeBuf] OF CHAR;
    END;
VAR
          Buffer:ARRAY [0. .Nrbufere] OF LanțBuf;
          Free:LINK; (*cap listă*)
          i:CARDINAL;
          chan:LINK;
          (*procedura privată de punere buffer în listă*)
PROCEDURE Ignorbuf (B:LINK);
BEGIN
      Buffer [B] .următ: = Free;
      Free: = B
END Ignorbuf;
PROCEDURE GetBuf (VAR B:LINK); (*procedură privată*)
BEGIN                                (*scoatere din listă*)
      IF Free<>NULL THEN
      B: = Free;
      Free: = Buffer[Free] .următ;
      ELSE
         WriteString („Buffer scos din listă"); WriteLn;
END GetBuf;
PROCEDURE depozit (ch:CHAR; chan:LINK);
BEGIN
      WITH Buffer [chan] DO
           BUFFER [ultim] . data [adrmax] : = ch;
           INC (adrmax);
           INC (contor);
           IF adrmax > SizeBuf THEN
               (*preluare buffere din listă*)
               GetBuf (Buffer [ultim] . următ);
               ultim:= Buffer [ultim] . următ;
               Buffer [ultim] . următ:= NULL;
               adrmax:=0
      END; (*IF*)
      END; (*WITH*)
END depozit;
PROCEDURE Extragere (VAR chan:LINK):CHAR;
VAR
      ch:CHAR; temp:LINK;
BEGIN
      ch:=OC;
WITH Buffer [chan] DO
      IF contor = 0 THEN
```


Scanned with OKEN Scanner

```
            WriteString („Nu se poate extrage din Buffer gol");
            WriteLn;
        ELSE
            ch:= data adrcurentă;
            DEC (contor);
            INC (adrcurentă);
        END; (*IF*)
END; (*WITH*)
IF Buffer [chan] . adrcurentă > SizeBuf THEN
                (*punere buffer în lista neînlănțuită*)
WITH Buffer [următ] . contor: = contor;
        Buffer [următ] . ultim: = ultim;
        Buffer [următ] . adrcurentă: = 0;
        Buffer [următ] . adrmax: = adrmax;
END (*WITH*)
        temp:= chan;
        chan:= Buffer [chan] . următ;
        IgnorBuf (temp);
END; (*IF*);
RETURN ch
END Extragere;
PROCEDURE OpenBuf (VAR chan:LINK);
BEGIN  (*calificare referințe antet buffer*)
        GetBuf (chan);
        WITH Buffer [chan] DO
        adrmax:= 0;
        adrcurent:= 0;
        următ:= NULL;
        ultim:= chan;
        cont:= 0;
        END; (*WITH*)
END OpenBuf;
BEGIN
        (*corpul modulului*)
        Free:= 1;
        FOR i:= 1 TO Nrbufere — 1 DO
                Buffer [i] . următ:= i+1
        END;
        Buffer [Nrbuffere] . următ:= NULL;
END LLBuf.
```

Înregistrări cu variante

Definirea unui tip RECORD presupune că toate cîmpurile care participă la definirea lui să fie formate din aceleași componente pentru toate înregistrările.

Dacă dorim ca definirea anumitor cîmpuri să fie diferită de la o înregistrare la alta în cadrul aceluiași tip (RECORD) vom spune că înregistrările respective au variante. Aceste variante se obțin prin in-

Scanned with OKEN Scanner

serarea uneia sau mai multor clauze CASE în secvența de declarare a cîmpurilor. Exemplificăm prin crearea de variante ale înregistrărilor ce conțin date din *FișaPersonală:*

```
TYPE DATA =
     RECORD
          ziua : [1..31];
          luna : ARRAY [1..10] OF CHAR;
          anul : CARDINAL;
     END;
TYPE StareCivilă = (căsătorit,celibatar);
TYPE FișaPersonală =
     RECORD
          Nume, Prenume:CHAR;
          DataNașterii : DATA;
          NrLegitimație: CARDINAL;
          CASE masculin: BOOLEAN OF
               TRUE : NumărLivret : CARDINAL
               FALSE: NumeAnterior: CHAR
     END;
          CASE:StareCivilă OF
          celibatar
          căsătorit: recăsătorit,divorțat,îndivorț:
          BOOLEAN
     END;
               NrdeCopii:CARDINAL;
END; (*RECORD*)
```

Se observă că în declararea tipului de date *FișaPersonală* au fost inserate două clauze CASE; una pentru declararea variantelor cîmpului cu identificatorul *„masculin"* și o clauză CASE pentru declararea variantelor unui cîmp fără identificator. Variantele cîmpului *„masculin"* în funcție de valorile „TRUE" sau „FALSE" ale tipului BOOLEAN sînt *NumărLivret* și *NumeAnterior,* care la rîndul lor conțin valori de tip CARDINAL și, respectiv, CHAR. Următoarea clauză CASE din *Fișa Personală,* în funcție de valorile variabilelor tipului enumerativ *Stare Civilă,* stabilește prezența variantelor de cîmpuri „Celibatar" sau „Căsătorit". La rîndul ei varianta *căsătorit* poate desemna valorile *„recăsătorit" „divorțat"* și *„îndivorț",* de tip BOOLEAN.

Cîmpurile care au variante sînt declarate deci prin intermediul clauzei CASE, ce respectă următoarea sintaxă:

```
Listă VarianteCîmpuri::=
     CASE[identificator]": "TipEtichetăCASE OF Variantă {"|" Variantă}
                    ELSE SecvențăCîmpuri
     END;
     Variantă::= ListăEticheteCASE ":"SecvențăCîmpuri
     ListăEticheteCASE::= EtichetăCASE {, EtichetăCASE}
     SecvențăCîmpuri::= ValoareCîmp {, ValoareCîmp}: TipDateCîmp ";"
```


Scanned with OKEN Scanner

TipEtichetăCASE reprezintă tipul de date al valorilor ce desemnează variantele de cîmpuri, fiind precedat facultativ de identificatorul acestora și de cuvîntul cheie „CASE". Urmează apoi cuvîntul cheie „OF" și enumerarea variantelor de cîmpuri separate prin simbolul „|". Fiecare variantă este stabilită de una sau mai multe etichete CASE *(ListăEticheteCASE)*, separate prin "," și urmate de simbolul ":" după care sînt înșiruite valorile curente ale variantelor de cîmpuri, separate prin ",". Aceste valori se mai numesc discriminatori între variantele posibile.

După enumerarea discriminatorilor, urmează semnul ":" și tipul de date al acestora *(TipDateCîmp)*.

Menționăm că fiecare cîmp poate avea la rîndul lui variante de subcîmpuri stabilite cu clauze CASE; totodată, instrucțiunea CASE nu trebuie să se confunde cu clauza CASE de declarare a variantelor de cîmpuri ale unor înregistrări.

Referindu-ne la utilizarea discriminatorilor în exemplul anterior, dacă dispunem de două înregistrări I1 și I2 din tipul *„FișaPersonală"*, cîmpul *I1.NumărLivret* există numai cînd *I1.masculin* = **TRUE**, iar cîmpul *I2.NumeAnterior* numai cînd *I2.masculin* = **FALSE**.

Deși folosirea înregistrărilor cu variante economisește importante resurse de memorare, utilizarea lor trebuie să se facă cu mare atenție căci selectarea valorilor unor cîmpuri poate crea confuzii.

CAPITOLUL 5

PROCEDURI

În practica programării este deja o obișnuință ca programele de mare complexitate să fie elaborate prin integrarea unor componente mai mici și ușor inteligibile, corespunzătoare structurii algoritmice și datelor problemei tratate. În această idee MODULA-2 asigură partiționarea programului la nivel de date, secvențe de control, proceduri și module.

5.1. Declararea procedurilor

Intuitiv, procedurile sînt un grup de secvențe de control și date ce pot fi apelate în diverse module ale programului. Procedurile grupează secvențe de control și date, dar pot conține și module locale. Putem

Scanned with OKEN Scanner

menționa cîteva deosebiri esențiale între proceduri și module. Modulele grupează atît secvențe de date și de control, dar și proceduri care pot fi exportate și deci importate de alte module. Totodată modulele nu pot fi nici activate și nici apelate, pe cînd o procedură poate fi atît apelată cît și activată în diverse module din cadrul programului.

O procedură este constituită din următoarele componente: antetul procedurii, blocul de identificatori și corpul procedurii. În MODULA-2 o procedură este definită printr-o declarație de procedură. Dacă la declararea unei variabile sau constante se specifică numele și tipul ei, la declararea unei proceduri se precizează numele, parametrii folosiți și — pentru proceduri funcționale (funcții) — tipul valorilor returnate. Declararea unei proceduri se plasează în cadrul unui modul-program după specificațiile de import și înainte de corpul modulului. Sintaxa declarației de procedură este:

DeclarațieProcedură::=
AntetProcedură";"
[*BlocIdentificatori*]
[*CorpProcedură*]
END *NumeProcedură*"."
AntetProcedură::= **PROCEDURE** *NumeProcedură ParametriFormali*
BlocIdentificatori::= {*DeclarațieConstantă*} | {*DeclarațieVariabilă*} |
{*DeclarațieTip*} | {*DeclarațieProcedură*} |
{*DeclarațieModulLocal*}
CorpProcedură::= **BEGIN** *SecvențăInstrucțiuni*

Nume Procedură este identificatorul care stabilește numele procedurii. *BlocIdentificatori* conține, ca și în cazul modulului-program, declarații de constante, declarații de variabile, declarații de tipuri, declarații de proceduri și — mai rar — chiar declarații de module locale.

Trebuie menționat că declarații de proceduri pot apare în blocul de identificatori al unui modul-program, al unui modul local, al unei proceduri sau al secțiunii de implementare a unui modul global. *CorpProcedură* este corpul procedurii și conține secvențe de instrucțiuni cuprinse între **BEGIN** și **END.** În argumentele acestor instrucțiuni pot apare diferite construcții sintactice obținute din identificatori: expresii, indicatori de componente selectate, indicatorii componentelor de tablouri sau înregistrări, indicatori de desemnare (variabile referite de pointeri). Aceeași observație este valabilă și pentru construcțiile sintactice din corpul unui modul.

Blocul de identificatori al unei proceduri împreună cu corpul procedurii se numește blocul procedurii.

La declararea unei proceduri parametrii se specifică prin intermediul parametrilor formali care țin locul celor desemnați pentru

Scanned with OKEN Scanner

a fi utilizați în procedură, numiți parametrii actuali. Corespondența între parametrii formali și cei actuali se stabilește în momentul apelului de procedură în ordinea existenței în antetul de procedură.

Parametrii formali se împart în două categorii: parametrii valorici și parametrii variabile. Sintaxa parametrilor formali este:

ParametriiFormali = "("[*SecțiunePF*{";"*SecțiunePF*}] ")"
[":" *TipReturnat*] ";"
SecțiunePF::= [**VAR**] *ListăIdentificatori* ":" *TipFormal*
TipFormal= [**ARRAY OF**] *IdentificatorTip*.

TipReturnat este tipul valorii returnate în cazul procedurilor funcționale. Parametrii, „variabile" care corespund parametrilor actuali sînt identificați de simbolul **VAR**, iar parametrii „valorici" sînt indicați prin absența simbolului **VAR**. Numele formale ale acestor parametrii sînt enumerate în *ListăIdentificatori* iar tipul fiecăruia este dat de *TipFormal*. Exemplificăm declararea procedurii *„Read"* din modulul *„Conversie"* care convertește în zecimal un caracter numeric codificat ASCII:

```
MODULE Conversie;
    FROM InOut IMPORT ReadChar;
    PROCEDURE Read(VAR x:CARDINAL);
        VAR i:CARDINAL; ch:CHAR;
    BEGIN i:=0;
        REPEAT ReadChar(ch);
        UNTIL (ch > "0") & (ch<="9");
        i:=10*i + (ORD(ch) - ORD("0"));
        x:=i;
    END Read;
    VAR y:=CARDINAL;
        x:=CARDINAL;
    BEGIN
        y:=5;
    Repeat Read(x); y: =y-1;
    UNTIL y=0;
END Conversie.
```

Se observă că lista parametrilor formali din antetul procedurii *„Read"* conține doar un parametru formal „x" de tip **CARDINAL** precedat de simbolul **VAR**.

Procedura *„CalculLungime"* din modulul *„LungimeSir"* redat mai jos conține în lista de parametrii formali parametrul „S" de tip **„ARRAY OF CHAR"**, iar tipul returnat este **„CARDINAL"**.

```
MODULE LungimeSir;
    CONST
        LgSir:=80;
    TYPE
        Sir:=ARRAY [0..LgSir] OF CHAR;
```


Scanned with OKEN Scanner

```
PROCEDURE CalculLungime (S:ARRAY OF CHAR):CARDINAL;
    VAR
        i:CARDINAL;
BEGIN
    FOR i=0 TO HIGH (S) DO
        IF S [i] =OC THEN RETURN i
        END (*IF*)
    END (*FOR*)
END CalculLungime.
```

5.2. Apelul unei proceduri

Apelul unei proceduri este mecanismul prin care o procedură poate fi activată ori de cîte ori este necesar în execuția programului. Regula esențială constă în aceea că la apelul unei proceduri trebuie să fie furnizați toți parametrii (actuali) care corespund parametrilor formali enumerați în definiția procedurii.

Sintaxa apelului de procedură este:

ApelProcedură:: = *NumeProcedură* [*Parametriactuali*]

Parametriactuali = „(„*expresie*{, *expresie*} ")"

După cum s-a mai amintit, parametrii pot fi *variabile* sau *valori*. Cînd parametrii sînt valori, modul de apel al unei proceduri se numește *apel prin valoare (call by value)*. Dacă parametrii sînt variabile declarate în antetul procedurii, modul de apel se numește *apel prin referință (call by reference)*. Există proceduri care sînt apelabile simultan prin valoare și prin referință.

La *apelul prin valoare* parametrii actuali sînt preluați de procedură fără ca aceștia să fi fost declarați sau asignați în exteriorul procedurii. Pentru toate procedurile apelate în acest mod se indică faptul că un parametru este o valoare prin simpla menționare a numelui de parametru urmat de tipul valorii respective. Parametrii actuali trebuie să fie combatibili la asignare cu tipul valorii declarat în antetul de procedură.

Apelul prin referință specifică faptul că parametrii actuali sînt variabile și corespund parametrilor formali, din definiția procedurii, care sînt tot variabile ce au fost declarate în exteriorul procedurii. Acest lucru este indicat prin prezența cuvîntului cheie VAR în antetul procedurii.

Vom compara apelul prin valoare cu apelul prin referință al procedurii *WriteBool* care este declarată o dată cu parametrul valoare și apoi cu parametrul variabilă:

```
MODULE ApelprinValoare;
FROM InOut IMPORT WriteString, WriteLn;
PROCEDURE WriteBool (p: BOOLEAN);
```


Scanned with OKEN Scanner

```
BEGIN
    IF p THEN
        WriteString (' Adevărat')
    ELSE
        WriteString('Fals')
    END;
    p:=TRUE;
END WriteBool;
VAR
    tf:BOOLEAN;
BEGIN
    tf:=FALSE;
    WriteBool(tf);WriteLn;
    WriteBool(tf);WriteLn;
END ApelprinValoare.
```

În urma execuției acestui modul vor fi tipărite cuvintele:

Adevărat
Adevărat

Aceasta înseamnă că apelul procedurii *WriteBool* modifică numai în corpul procedurii parametrul menționat în antetul procedurii, valoarea lui rămînînd neschimbată în exteriorul ei.

Apelîndu-se procedura *WriteBool* prin referință, ca în exemplul următor, vom observa că parametrul actual „tf" exterior procedurii s-a modificat la apelul procedurii, astfel că execuția modulului *VarParam* va tipări cuvintele:

Fals
Adevărat

Pentru a ilustra acest mod de apel vom relua exemplul anterior și vom arăta apelul prin referință al procedurii *WriteBool*:

```
MODULE VarParam;
    FROM InOut IMPORT WriteString, WriteLn;
    PROCEDURE WriteBool (VAR p: BOOLEAN);
    BEGIN
        IF p THEN
            WriteString ('Adevărat')
        ELSE
            WriteString ('Fals')
        END
    END WriteBool;
    VAR
        tf:BOOLEAN; (*variabila tf va deveni parametru actual al procedurii
                                    WriteBool*)
    BEGIN
        tf: = FALSE; (*Asignare a parametrului cu o valoare*)
        WriteBool (tf); WriteLn;
        WriteBool (tf); WriteLn;
END VarParam.
```


Scanned with OKEN Scanner

Utilizarea apelului prin referință impune respectarea următoarelor condiții:

— la apelul procedurii parametrii actuali nu pot fi expresii și nici constante. Acest lucru este posibil numai dacă asupra lor s-a efectuat o asignare prealabilă (ca în exemplul precedent);

— parametrii formali din definiția procedurii și parametrii actuali trebuie să fie de același tip.

Procedurile fără parametrii pot fi apelate cu lista de parametrii vidă. De exemplu putem face următoarea asignare:

Val: = *expresie*()

unde *expresie* este o procedură care a fost declarată fără parametri.

5.3. Instrucțiunea RETURN

Din punctul de vedere al efectului unei proceduri acestea se pot clasifica în proceduri ordinare și, respectiv, funcționale (funcții).

Procedurile ordinare folosesc la intrare parametrii dați; ca urmare a unor secvențe de instrucțiuni ele returnează sau modifică un anumit număr de variabile. Execuția unei proceduri ordinare produce un număr de valori corespunzătoare variabilelor declarate în blocul procedurii, sau modifică valorile unor variabile ce nu sînt declarate în blocul procedurii dar sînt vizibile în acesta.

Procedurile funcționale sînt folosite cu un scop explicit de a obține un număr de rezultate. Aceste rezultate (valori) sînt de fapt valori ce pot fi utilizate în calculul unor expresii. De exemplu, procedura „cos" calculează cosinusul trigonometric al unei valori reale și returnează tot o valoare reală care poate fi utilizată în calculul următoarei expresii:

d: = a**2 + b**2 + 2.0*a*b*(—cos(alfa)).

Deducem regula următoare:

— procedurile ordinare nu pot fi utilizate în expresii;

— procedurile funcționale nu pot fi utilizate decît în calculul unor expresii.

Existența unei proceduri funcționale se menționează în declararea procedurii prin indicarea tipului valorii returnate în antetul de procedură la *TipReturnat*.

De exemplu, antetul declarării procedurii „cos" este:

PROCEDURE cos (alfa: REAL):REAL;

Trebuie reținută regula conform căreia tipul rezultat nu poate fi RECORD, ARRAY sau SET. Rezultatele sînt returnate de procedurile funcționale prin utilizarea instrucțiunii RETURN. Instrucțiunea

Scanned with OKEN Scanner

RETURN într-o procedură funcțională este urmată de o expresie, a cărei valoare este pasată înapoi la apelant în momentul în care este întîlnită instrucțiunea RETURN. Dacă există mai multe instrucțiuni RETURN într-o procedură numai una din ele va fi executată la fiecare apel al procedurii.

În procedurile obișnuite instrucțiunea RETURN este utilizată fără a fi urmată de o expresie și indică pur și simplu terminarea procedurii. În cazul absenței instrucțiunii RETURN dintr-o procedură ordinară sfîrșitul procedurii este marcat de ultima instrucțiune executată din procedură. Instrucțiunea RETURN mai este folosită și în cadrul corpului unui modul pentru a desemna terminarea execuției acestuia. Acest aspect va fi discutat în capitolul 6.

Acum vom da două exemple de utilizare a instrucțiunii RETURN pentru proceduri ordinare și proceduri funcționale. Primul program va conține o procedură ordinară de calcul a lui $n!$:

```
MODULE Obișnuit;
    FROM InOut IMPORT WriteCard, WriteLn;
    (*procedură ordinară*)
    PROCEDURE Factorial (n:CARDINAL; VAR factorial:CARDINAL);
    BEGIN
        IF n>12 THEN factorial:=0;RETURN END;
        factorial:=1;
        FOR n=n TO 1 BY -1 DO factorial:= factorial *n END;
        END Factorial;
        VAR
            i,f:CARDINAL;
    BEGIN
            FOR i:=     1 TO 12 DO
              Factorial (i, f);
              WriteCard (i,3);
              WriteCard (f,20);
              WriteLn;
    END
END Obișnuit.
```

Execuția acestui program va produce tipărirea pe suportul de ieșire a tuturor numerelor naturale de la 1 la 11, alături de factorialele lor.

Același lucru îl putem obține folosind o procedură funcțională pentru calculul lui $n!$:

```
MODULE Funcțional;
    FROM InOut IMPORT WriteCard, WriteLn;
    (*procedură funcțională*)
    PROCEDURE Factorial (n:CARDINAL):CARDINAL;
    VAR factorial: CARDINAL;
    BEGIN
        IF n > 100 THEN RETURN 0 END;
        factorial:=1;
```


Scanned with OKEN Scanner

```
    FOR n:=n TO 1 BY -1 DO
            factorial: = factorial * n
    END;
    RETURN factorial; (*Aici este returnată valoarea factorial*)
END Factorial;
    VAR
        i:CARDINAL;
BEGIN
    FOR i:=1 TO 10 DO
            WriteCard (i,3);
            WriteCard (Factorial (i), 20);
            WriteLn
END (*FOR*)
END Funcțional.
```

MODULA-2 pune la dispoziția programatorului un set de funcții standard, care pot fi apelate fără a mai fi declarate în cadrul programului. Acestea sînt:

funcția	*Acțiunea*
ABS(x)	Calculează valoarea absolută a lui x; tipul valorii rezultate = tipul argumentului.
CAP(x)	Pentru argument caracter cu literă mică rezultatul este litera mare corespunzătoare. Pentru argument literă mare rezultatul este aceeași literă.
CHR(x)	Asociază numărului ordinal x caracterul care are codul octal ASCII x CHR(x) = VAL(CHAR, x).
FLOAT (x)	Asociază numărului cardinal x o valoare corespunzătoare de tip REAL.
HIGH(A)	Asociază tabloului A numărul de elemente al tabloului.
ODD(x)	Asociază unui număr întreg valoarea booleană TRUE, dacă x este impar, și FALSE, dacă x este par.
ORD(n)	Asociază un număr ordinal elementului x al unuia din tipurile CHAR, INTEGER, CARDINAL sau de enumerație.
TRUNC(x)	Asociază unui număr real pozitiv partea întreagă.
VAL(T,x)	Asociază celui de-al n-lea element al tipului T valoarea lui. T este tip enumerativ, CHAR, INTEGER, sau CARDINAL. **VAL**(T, **ORD**(x)) = x dacă x este de tipul T.
DEC(x)	$x:=x-1$
DEC(x, n)	$x:=x-n$
EXCL(s,i)	$s:=s-\{i\}$
HALT	Termină forțat execuția programului.
INC(x)	$x:=x+1$
INC(x, n)	$x:=x+n$
INCL(s,i)	$s:=s+\{i\}$

Parametrii funcțiilor INC și DEC pot fi și de tip enumerativ și caracter, caz în care x este înlocuit cu cel de-al n-lea element succesiv sau anterior.

În capitolul 7 vor fi amintite și funcțiile NEW, DISPOSE, TSIZE, ALLOCATE, DEALLOCATE.

Scanned with OKEN Scanner

5.4. Vizibilitatea identificatorilor

În MODULA-2 se utilizează identificatori pentru a specifica constante, variabile, proceduri, module, tipuri de date.

În multe cazuri se poate folosi același identificator pentru diferite elemente din cadrul programului. Pe de altă parte, programatorul are posibilitatea de a folosi pe porțiuni limitate identificatori care nu au aceeași semnificație în afara acestor porțiuni. Pentru evitarea ambiguităților sau a folosirii lor incorecte în ingineria programării se utilizează noțiunea de domeniu de vizibilitate al identificatorilor unei proceduri. Acesta reprezintă zona din program unde un identificator este valabil și accesibil. Regulile de definire a domeniului unui identificator sînt:

- domeniul unui identificator este procedura în care a fost declarat;
- dacă un identificator i declarat în procedura P este redeclarat în procedura Q conținută în procedura P, atunci procedura Q și toate procedurile cuprinse în Q nu fac parte din domeniul lui i;
- domeniul identificatorilor standard al limbajului este întregul program.

Precizăm că procedura Q este conținută în procedura P dacă Q este declarată în blocul de identificatori ai procedurii P.

Vom exemplifica modul de manifestare a vizibilității variabilelor în blocul unei proceduri în cazul următorului program ce conține procedura LOCAL conținută în procedura GLOBAL:

```
MODULE VarIdent;
    IMPORT.....
    VAR n: CARDINAL;
    VAR m: REAL;
    PROCEDURE GLOBAL;
    VAR p: INTEGER;
    VAR n: BOOLEAN;
    PROCEDURE LOCAL
    VAR p: CARDINAL;
    BEGIN (*LOCAL*)
    (*Aici sînt vizibili p: CARDINAL, n: BOOLEAN, m:REAL*)
    END LOCAL;
    BEGIN (*GLOBAL*)
    (*Aici sînt vizibili n: BOOLEAN, m: REAL, p: INTEGER*)
    END GLOBAL.
END VarIdent.
```

Se pot face următoarele observații cu privire la vizibilitatea identificatorilor folosiți în blocul unei proceduri:

— toți identificatorii declarați în exteriorul unei proceduri (modulul sau procedura care o conține) sînt vizibili în blocul procedurii dacă numele lor sînt diferite de numele identificatorilor creați în blocul de identificatori al procedurii;

Scanned with OKEN Scanner

— dacă există vreun conflict (același nume) între identificatori sînt vizibili numai cei declarați în blocul de identificatori;

— în afara blocului unei proceduri toți identificatorii declarați în blocul de identificatori al acesteia nu sînt vizibili.

Reamintim că blocul unei proceduri este format din blocul de identificatori și corpul procedurii.

5.5. Recursivitatea procedurilor

Procedurile pot fi nu numai apelate și activate de alte proceduri dar — mai mult — se pot apela pe ele însele. Această autoactivare se numește recursivitate.

Recursivitatea se prezintă sub două forme:

— recursivitatea directă, bazată pe apelul procedurii de către ea însăși;

— recursivitatea indirectă, bazată pe apelul unei alte proceduri care după terminare apelează procedura apelantă.

Vom ilustra primul caz prin varianta recursivă a calculului lui $n!$ pentru $1 \leqslant n \leqslant 10$:

```
MODULE Rfact;
    FROM InOut IMPORT WriteCard, WriteLn;
    PROCEDURE RecursiveFact (n:CARDINAL) :CARDINAL;
    BEGIN
        IF n:=0 THEN RETURN 1
        ELSIF n > 10 THEN RETURN (*eroare*)
        ELSE RETURN n * RecursiveFact (n-1)
        END
        END RecursiveFact;
        VAR
        i:CARDINAL;
        BEGIN
        FOR i= 0 TO 10 DO
            WriteCard (i,5);
            WriteCard (RecursiveFact (i),100);
            WriteLn;
        END
END Rfact.
```

Din punctul de vedere al vitezei de calcul varianta nerecursivă este mai rapidă decît varianta recursivă. Totuși, pentru tratarea mai sugestivă a unor clase de algoritmi este necesară folosirea recursivității.

Scanned with OKEN Scanner

CAPITOLUL 6

MODULE (Abstracții de date)

Modulele definesc structura organizațională a programelor scrise în MODULA-2. Rolul major al modulelor este de a crea partiționarea programelor, fiecare modul conținînd colecții de constante, tipuri de date, instrucțiuni și proceduri. Structura creată de module face ca un program să fie o ierarhie de module.

5.1. Module locale — vizibilitate

Reamintim că modulele locale sînt acelea care pot fi incluse în alte module. Fiecare modul local stabilește un domeniu de vizibilitate a identificatorilor lui. Regulile de stabilire a domeniului de vizibilitate sînt următoarele:

- dacă un identificator este vizibil în exteriorul unui modul local (modulul care îl conține) și nu apare în lista de „EXPORT" a acestuia va fi vizibil în interiorul lui dacă și numai dacă este menționat în lista de „IMPORT". Presupunînd de exemplu că un modul program MP importă servicii (date, proceduri) dintr-un modul global sau conține declarații de identificatori (servicii), aceștia sînt vizibili într-un modul local ML al lui MP dacă sînt prezenți în lista de „IMPORT" a lui ML;

* • cînd un identificator este vizibil în interiorul unui modul local și nu apare în lista de „IMPORT" a acestui modul, el este vizibil și în exteriorul lui (modulul care îl conține) dacă și numai dacă este prezent și în lista de „EXPORT". Presupunînd de exemplu că un modul program MP dorește să folosească anumite servicii de la un modul local ML care este conținut în MP trebuie ca acestea să figureze în lista de „EXPORT"a lui ML, caz în care ML și MP vor fi în același domeniu de vizibilitate în raport cu aceste servicii.

Listele de export sînt prezente explicit numai în module locale și implicit în secțiunea de definire a modulelor globale. Sintaxa unei liste este:

ListaExport::=**EXPORT** [**QUALIFIED**] *ListaIdentificatori*;

Cuvîntul cheie QUALIFIED în lista de „EXPORT" stabilește că toate elementele (serviciile) exportate pot fi utilizate (calificate) în domeniul de vizibilitate din exteriorul modulului, folosind indicatori de componente selectate, fapt care evită conflictul între identificatori cu același nume dar provenind din module diferite. Omiterea cuvîntului

QUALIFIED obligă programatorul să verifice ca numele respective să nu mai fie deja folosite în domeniul lor de vizibilitate. Atributul QUALIFIED se aplică tuturor exporturilor, întrucît modulele pot avea o singură listă de export. Trebuie remarcat că și atributul QUALIFIED este implicit în cazul modulelor globale, deoarece acestea pot fi realizate de programatori diferiți care, în mod normal, nu știu ce identificatori au folosit ceilalți programatori. În cazul modulelor locale atributul QUALIFIED are un caracter opțional și reflectă o facilitate sintactică de evitare a omonimiei identificatorilor. Sintaxa declarării unui modul local este similară declarării unui modul program cu excepția prezenței listei de export:

```
DeclarațieModulLocal::= MODULE Identificator [prioritate] ";"
                        {listaImport ";"}
                        listaExport ";"
                        BlocIdentificatori ";"
                        [BEGIN
                            CorpdeModulLocal]
                        END Identificator "."

BlocIdentificatori::=
        {DeclarațieConstantă} | {DeclarațieVariabilă} |
        {DeclarațieTip}       | {DeclarațieProcedură} |
        {DeclarațieModulLocal}
```

Identificatorii prezenți în lista de export (constante, variabile, tipuri, proceduri) trebuie obligatoriu să fie declarați în modulul care îi exportă. Exportul lor îi face cunoscuți în blocul care conține declararea modulului ca și cînd ei ar fi fost declarați în acest bloc. Există un amendament referitor la exportul constantelor și tipurilor, în sensul că acestea nu pot fi utilizate decît în declarațiile ce urmează modulului exportator. Dacă se exportă obiecte structurate (înregistrări) sau tipuri de înregistrări, atunci calificatorii tuturor acestor componente sînt automat exportați. Dacă un tip enumerativ este exportat, atunci și indicatorii constantelor lui sînt automat exportați. Identificatorii specificați în listele de import desemnează obiecte care au fost declarate în afara modulului, iar acestea pot fi utilizate în modul ca și cînd ar fi fost declarate în blocul de identificatori al acestuia. Identificatorii standard ai limbajului sînt automat importați, prezența lor nefiind necesară în listele de import și export.

Dacă facem o comparație între module locale și proceduri din punctul de vedere al vizibilității, în blocul unei proceduri sînt vizibili (fără a fi necesar importul lor ca în cazul modulelor locale) toți identificatorii vizibili în blocul procedurii sau modulului în care aceasta este declarată.

Identificatorii locali ai unei proceduri nu sînt vizibili în exteriorul procedurii (blocul modulului sau procedurii ce o conține), pe cînd iden-

Scanned with OKEN Scanner

tificatorii unui modul local pot fi făcuți vizibili în exteriorul acestuia prin menționarea lor în lista de export a modulului.

Exemplu: Următorul program conține un modul local numit *„ascuns"* și trei variabile, *q, r* și *s*, ce nu pot fi folosite de modulul program numit *„Ambiant"*, întrucît nu sînt menționate într-o listă de export. Modulul *„Ambiant"* conține trei variabile *p, q, r* diferite de cele din modulul local; nici ele nu pot fi folosite în modulul local, deoarece nu există o listă care să le importe:

```
MODULE Ambiant;
VAR
    p, q, r: CARDINAL;
    MODULE ascuns ; (*modul local*)
    VAR
        q,r,s: CARDINAL;
    BEGIN
        (*q,r,s (locale) definite în modulul ascuns sînt vizibile; p,q,v definite
        în modulul program Ambiant nu sînt vizibile în modulul ascuns*)
    END ascuns;
BEGIN
        (*variabilele p,q,r sînt vizibile, iar variabilele q, r și s definite în modulul
        ascuns nu sînt vizibile*)
END Ambiant.
```

Identificatorii vizibili numai în blocul unui modul sau în proceduri și invizibili în afara modulului sau procedurii se mai numesc identificatori locali.

Următorul exemplu arată cum o listă de export cu atributul QUALIFIED evită coincidența de nume de variabile ale unui modul local și ale unui modul program:

```
MODULE coliziune;
VAR
    portocale, banane: CARDINAL;
    MODULE fructe ; (*modul local*)
    EXPORT QUALIFIED portocale, grapefruit;
    VAR
        portocale, grapefruit, mandarine, banane:CARDINAL;
    BEGIN (*corpul modulului fructe*)
        (*În acest modul sînt vizibile doar variabilele definite în modulul lo-
        cal fructe*)
    END fructe;
BEGIN
    (*Aici sînt vizibile variabilele portocale, banane din modulul program și va-
    riabilele selectate fructe, portocale, fructe, grapefruit din modulul fructe*)
    portocale:= 20; (*portocale din modulul program*)
    fructe.portocale:= 100; (*din modulul local fructe*)
END coliziune.
```

Se poate observa că prezența exportului cu atributul QUALIFIED a evitat coincidența de nume a *„portocalelor"* provenite din module diferite, prin folosirea de componente selectate.

Scanned with OKEN Scanner

O restricție sintactică importantă a listelor de import sau export este că acestea pot conține numai identificatori, expresiile, componentele selectate, indicatorii componentelor de tablouri sau înregistrări, precum și indicatorii de desemnare (variabile referite de pointeri) nefiind permiși.

6.2. Corpuri de module locale

Corpul unui modul este secvența de instrucțiuni limitate de cuvintele **BEGIN** și **END** ce urmează după declarațiile din modul cuprinse în blocul de identificatori. Exemplele date anterior nu descriu secvențele conținute în corpurile de module. Trebuie remarcat că secvențele de instrucțiuni din modulele locale se execută înaintea secvențelor de instrucțiuni din modulul program. Dacă sînt mai multe module locale cuprinse într-un anumit modul, secvențele de instrucțiuni din modulele locale sînt executate în ordinea declarării lor în program și înaintea secvenței de instrucțiuni din modulul care le conține.

Exemplu:

```
MODULE ordine;
FROM InOut IMPORT WriteString, WriteLn;
    MODULE A;
    IMPORT WriteString, WriteLn;
    BEGIN (*Corpul de instrucțiuni al modulului A*)
          WriteString ('Execuția de instrucțiuni')
          WriteString ('din modulul A');
          WriteLn
    END A;
    MODULE B;
    IMPORT WriteString, WriteLn;
    BEGIN (*corpul de instrucțiuni al modulului B*)
          WriteString (*Execuția de instrucțiuni*);
          WriteString (*din modulul B*);
          WriteLn
    END B;
    BEGIN (*corpul de modul program*)
          WriteString (*Execuția secvenței de instrucțiuni*);
          WriteString (*din modulul program*);
          WriteLn;
END ordine.
```

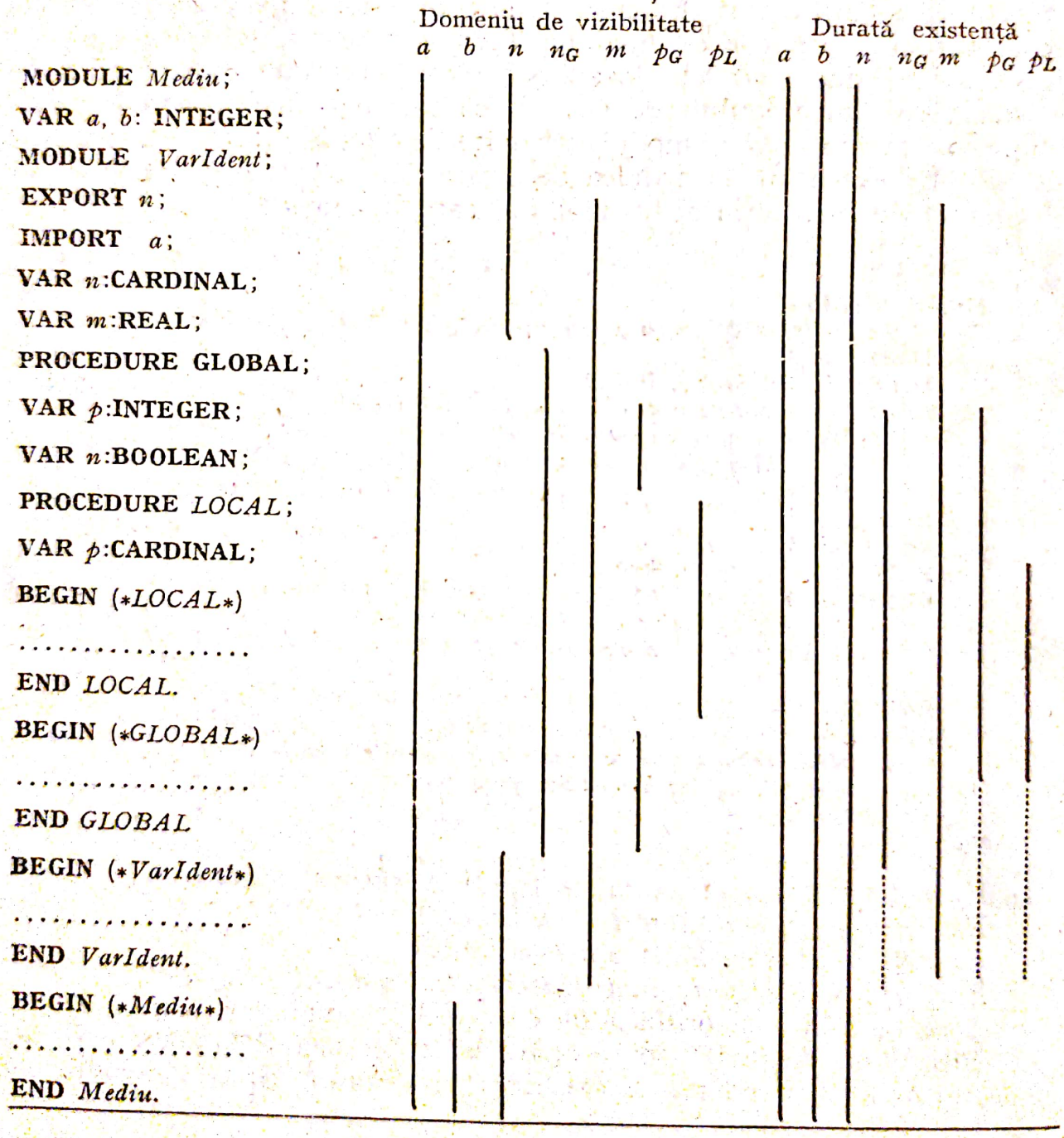
Execuția acestui program va tipări următorul text:
Execuția de instrucțiuni din modulul A
Execuția de instrucțiuni din modulul B
Execuția secvenței de instrucțiuni din modulul program
Aceasta indică și ordinea de execuție a modulelor.
Într-un modul local poate apare instrucțiunea RETURN care indică terminarea imediată a execuției modulului. Spre deosebire de

Scanned with OKEN Scanner

proceduri, instrucțiunea RETURN din corpul unui modul nu poate avea argumente.

Ordinea de execuție a modulelor locale, precum și apelurile de proceduri au consecințe imediate asupra duratei de existență a variabilelor din cadrul modulului program. În acest sens, se pot distinge, pe lîngă caracteristicile legate de tipul variabilelor, și cele legate de durata lor de existență.

Vom sugera în exemplul următor o comparare a domeniului de vizibilitate a identificatorilor definiți atît în blocuri de module locale, cît și de proceduri, cu durata lor de existență:



Domeniul de vizibilitate a variabilei *a* este tot blocul modulului *„Mediu"*, deoarece apare în blocul de identificatori ai acestuia. Fiind importată de modulul *„VarIdent"*, este vizibilă și în blocul lui (și deci și în blocul procedurilor LOCAL și GLOBAL), spre deosebire de variabila *b* care, nefiind importată, e vizibilă numai în modulul *„Mediu"*. Variabila *b*, deși nu este vizibilă în blocul lui *„VarIdent"*, va exista pe toată durata execuției lui.

În exemplul dat s-a notat cu n_G variabila definită în procedura GLOBAL spre a o distinge de variabila *n* (deși au același identificator) definită în modulul *„VarIdent"*. Totodată, p_G și p_L reprezintă respectiv variabila *p* definită în procedura GLOBAL și variabila *p* definită în procedura LOCAL.

Cu linie punctată s-a marcat prelungirea posibilă a duratei de existență a variabilelor ca urmare a apelurilor de proceduri în care acestea (variabilele) sînt declarate.

Variabilele statice durează tot timpul execuției programului, pe cînd variabilele dinamice există o perioadă mai scurtă (de exemplu, cît timp durează apelul unei proceduri sau execuția unui model local). La fiecare execuție a unei proceduri se creează automat o copie a variabilelor locale, procedurii sau modulului local în al cărui corp se apelează procedura.

Datorită acestui fapt trebuie reținut, ca un element referitor la economia programului, că spațiul de memorie alocat variabilelor dinamice crește proporțional cu numărul de apeluri al procedurilor în care acestea sînt definite.

Vom ilustra cîteva caracteristici ale vizibilității și ale temporalității variabilelor în programul următor de căutare a numerelor prime după metoda ciurului lui Eratostene. Modulul-program *„NrPrime"* conține modulul local *„WriteMod"* cu proceduri de editare și variabile statice care țin evidența liniilor și coloanelor din tabelul numerelor prime găsite:

```
MODULE NrPrime;
FROM InOut IMPORT WriteCard, WriteLn, WriteString;
     MODULE WriteMod; (*modul local*)
     IMPORT WriteCard, WriteLn;
     (*în modulul program sînt vizibile proceduri importate din modulul global
     InOut; prezența lor în lista de import a modulului local le face vizibile și
     în WriteMod*)
     EXPORT WRITECARD;
     (*prezența în lista de export a lui WRITECARD face*)
     (*vizibila aceasta procedură în modulul program*)
```


Scanned with OKEN Scanner

```
        VAR
            linie, col. : CARDINAL;
            (*variabile statice care deşi sînt locale*)
            (*vor fi utilizate pe toată durata execuţiei*)
            (*programului*)
            PROCEDURE WRITECARD (x:CARDINAL);
            BEGIN
                    WriteCard (x, 7);
                    INC (col);
                    IF col = 10 THEN
                            col: = 0;
                            WRITELN;
                    END;
            END WRITECARD;
            PROCEDURE WRITELN;
            VAR
                    i : CARDINAL; (*variabila dinamică a cărei durată e con-
                                    diţionată de numărul de apeluri al pro-
                                    cedurii*)
            BEGIN
                    INC (linie):;
                    WriteLn;
                    IF linie = (PageLenghth - 4) THEN
                            FOR i:= 1 to 7 DO WRITELN END;
                    ELSIF linie = PageLenght THEN
                            linie:=0;
                    END;
            END WRITELN;
        BEGIN (corpul modulului local WriteMod*)
              col:= 0;
            linie:= 0;
           WRITELN;
        END WriteMod;
CONST
            Last = 20000;
VAR
        flags : ARRAY [0..Last] OF BOOLEAN; (*variabilă statică*)
        suspect, depistat, prim: CARDINAL; (variabile statice*)
BEGIN (*corpul modulului program — alg. Eratostene*)
        FOR suspect := 0 TO Last DO
        flags [suspect] := TRUE
        END (*FOR*);
        FOR suspect := 0 TO Last DO (*cautare*)
            IF flags [suspect] THEN (*găsit număr prim*)
               prim:= suspect + suspect + 3;
               WRITECARD (prim);
               depistat:= prim + suspect;
               WHILE depistat = Last DO
                   (*eliminare multipli de număr prim*)
               flag[depistat] = FALSE
               depistat:= depistat + prim;
```


Scanned with OKEN Scanner

```
            END (*while*)
          END (*if*)
        END (*for*)
        WriteLn
    END NrPrime.
```

În acest exemplu variabilele „linie" și „col" — deși private în modulul „*WriteMod* — sînt permanent utilizate pe toată durata execuției programului, spre deosebire de *i* care este variabila dinamică.

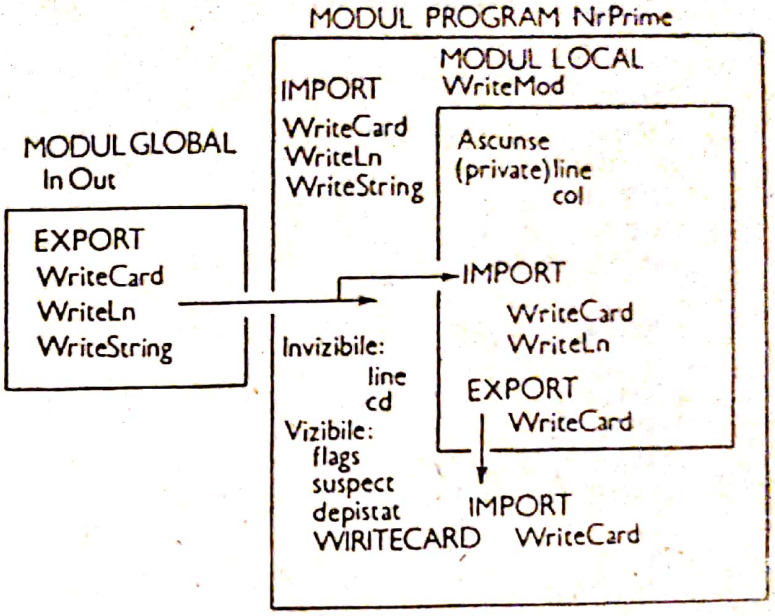


Fig. II.3. Prin importul în modulul-program a procedurilor WriteCard, WriteLn, și WriteString, domeniul lor de vizibilitate s-a extins în blocul programului. Pentru a fi vizibile și în modulul local WriteMod aceste proceduri pot fi importate de către modulul menționat fără a specifica modulul exportator. De asemenea, pentru a putea intra în domeniul de vizibilitate al modulului-program, procedura WriteCard trebuie exportată de modulul local WriteMod.

Ambianțe de module locale.

Pentru a pune în evidență unele proprietăți ale modulelor locale și ale procedurilor este indicat să definim noțiunea de context ambiant.

Contextul ambiant al unui modul local este modulul sau procedura în care a fost declarat modulul local. Contextul ambiant al modulului program sau al modulelor globale este mediul de programare în care acestea sînt dezvoltate.

Dacă modulul A este contextul ambiant al modulului B se mai spune că modulul B este cuibărit în modulul A.

Relația de cuibărire nu este tranzitivă: dacă modulul A este cuibărit în modulul B iar B este cuibărit în modulul A nu se poate spune că D este cuibărit în A.

Faptul că relația aceasta nu este tranzitivă are atît consecințe sintactice cît și semantice.

Sintactic, dacă modulul B este declarat în blocul de identificatori al modulului A atunci din faptul că D este declarat în blocul de identificatori al lui B nu putem spune că D este declarat în blocul lui A.

Scanned with OKEN Scanner

Semantic, deoarece un modul local poate importa și exporta numai din și respectiv în contextul ambiant, dacă un modul este cuibărit în alt modul cele două module au contexte diferite.

În cazul că un modul local dorește să folosească servicii dintr-un modul global, acest lucru este posibil numai după ce serviciul respectiv a fost importat în contextul ambiant al modulului local (dacă contextul ambiant e tot un modul) și apoi importat de modulul local însuși.

În exemplul următor modulul B este contextul ambiant al modulului D iar modulul A este contextul ambiant al modulului B și al lui C.

```
MODULE A;
FROM InOut IMPORT WriteLn;
MODULE B;
EXPORT P;
IMPORT WriteLn;
PROCEDURE P;
.....................
BEGIN (*corpul procedurii*)
.....................
END P.
MODULE
IMPORT WriteLn;
BEGIN (*corpul modulului D*)
.....................
END D.
MODULE c;
IMPORT P;
BEGIN (*corpul modulului c*)
.....................
END c.
BEGIN P; (*corpul modulului A*)
.....................
END A.
```

În acest exemplu procedura WriteLn din modulul global InOut pentru a fi importată de modulul D trebuie mai întîi importată de contextul ambiant (modulul B) din modulul A (contextul ambiant al lui B).

Modulul B exportă procedura P în contextul lui ambiant, B și C fiind în același context ambiant, procedura P poate fi importată și de modulul C.

Modulele locale creează nivele de cuibărire. În exemplul dat, modulele B și C au același nivel de cuibărire (nivelul A) pe cînd modulul D are un nivel inferior de cuibărire (nivelul B).

Similar modulelor locale contextul ambiant al unei proceduri poate fi o procedură sau un modul (local, global sau program).

Procedurile pot fi și ele cuibărite deci în proceduri sau module locale și au și ele o interfață care le separă de contextul ambiant. Interfața

Scanned with OKEN Scanner

modulelor locale este formată din listele de import și export în cazul procedurilor antetul de procedură.

Aici trebuie să remarcăm că identificatorii locali (definiți în blocul de identificatori ai procedurii) nu pot fi vizibili în contextul ambiant al procedurii pe cînd identificatorii unui modul local pot fi vizibili în contextul ambiant al acestuia prin prezența lor în lista export. Altă deosebire este că identificatorii vizibili în contextul ambiant al procedurii sînt vizibili în blocul procedurii dacă au nume diferite de identificatorii locali ai acesteia (la module locale identificatorii vizibili în contextul ambiant sînt vizibili în blocul modulului prin prezența lor în listele de import).

În exemplul următor procedura A este contextul ambiant al modulului B iar modulul B este contextul ambiant al procedurii C.

```
MODULE Cuib;
FROM inOut IMPORT Write, WriteLn;
VAR
    c. CHAR;
PROCEDURE A;
VAR
    b:CHAR;
MODULE B;
IMPORT c, Write;
VAR
    a:CHAR;
PROCEDURE C (a:CHAR):CHAR;
BEGIN (*corpul procedurii C*)
    Write (c);
    INC (a);
    c: = a;
    INC (c);
    RETURN c
END C.
BEGIN (*corpul modulului B*)
    a: = 'M'
    Write (c(a));
    c: = a;
    DEC (c);
END B.
BEGIN (*corpul procedurii A*)
    Write ('D')
END A.
BEGIN
    A;
    Write ('n');
    DEC (c);
    Write (c);
    Write ('A');
    WriteLn;
END Cuib.
```


Scanned with OKEN Scanner

În acest exemplu variabila c definită în contextul ambiant al procedurii A va fi automat vizibilă și în blocul ei.
Însă în procedura A este cuibărit modulul B deci A este context ambiant pentru B. Variabila c va fi vizibilă în blocul lui B numai prin prezența ei în lista de import a modulului local B.

Pentru a surprinde altă deosebire între module și proceduri vom defini noțiunea de context de execuție ca fiind locul din program unde o instrucțiune, o procedură sau un modul se execută. Contextul de execuție al unei proceduri este variabil (există un loc de declarare și unul sau mai multe locuri de execuție); ea poate fi apelată cu diferite valori ale parametrilor ei în mai multe locuri din corpul unui modul sau proceduri.

Un modul local nu se poate executa decît în locul în care a fost declarat (contextul ambiant este același cu contextul de execuție).

Pentru modulul local variabilele modulului sînt create cînd contextul ambiant (procedura sau modulul) începe execuția. Dacă modulul cuibărit într-o procedură conține corp de modul, corpul modulului se execută înaintea corpului procedurii.

6.3. Module globale

În procesul de partiționare a programului în module globale sînt esențiale atît ascunderea unor detalii constructive de obiecte (tipuri, proceduri etc.), cît și specializarea. De exemplu, poate fi necesară existența unui modul specializat în operații de intrare-ieșire, altul în funcții grafice etc. Componentele acestor module pot fi utilizate în program integral, dar de cele mai multe ori se folosesc numai anumite proceduri conținute în ele. Modulele definite în exteriorul modulului-program se numesc module globale, iar accesul la procedurile sau datele acestora se face ignorîndu-se sau nu anumite detalii constructive. Pe de altă parte, modulul global poate ascunde sau face vizibile în exteriorul lui anumite detalii. Proprietățile modulelor globale determină ca ele să constituie părți separate ale programului și le oferă protecție la accesul din exterior, făcînd posibilă modificarea modulelor exportatoare fără modificarea sau compilarea modulului importator. Acest lucru este foarte important în cazul în care anumite module sînt dezvoltate de diferiți utilizatori sau depind de sistemul de operare sub care este implementat limbajul. Specificarea separată a componentelor diverselor obiecte de detaliile lor se face prin împărțirea modulului global în două secțiuni — *de definire* și *de implementare* — și prin enumerarea în secțiunea de definire a obiectelor care sînt vizibile în exteriorul modulului global.

Scanned with OKEN Scanner

Secțiunea de definire anunță serviciile modulului și conține declararea obiectelor exportate.

Secțiunea de implementare este o parte ascunsă (invizibilă în exteriorul modulului) și realizează propriu-zis serviciile modulului.

Cele două secțiuni ale modulului global se pot dezvolta separat, încărca în fișiere distincte și compila separat.

Secțiunea de definire a modulelor globale.

Definirea modulelor globale are două roluri: de a anunța facilitățile modulului și de a descrie interfața modulului cu mediul de programare MODULA-2.

Sintaxa secțiunii de definire a unui modul global este:

SecțiuneDefinireModulGlobal::=
    DEFINITION MODULE *NumeModul*";"
        {*ListăImport*}
    Definiție
    END *NumeModul*"."
ListăImport::= [**FROM** *Ident*] **IMPORT** *ListaIdentificatori*;
Definiție::= {*DeclarațieConstantă*;} | {*DeclarațieVariabilă* ";"} |
    {*DeclarațieIncompletăTip*"} | ";"{ {*AntetProcedură* ";"} |

NumeModul este identificatorul de nume al modulului global.

ListaIdentificatori conține constante, variabile, proceduri necesare în secțiunea de definire a modulului, fiind importate din alte module globale.

Ident este identificatorul de nume al modulului global de unde se face importul. În absența cuvîntului cheie „**FROM**", *ListăIdentificatori* va conține numai nume de module globale, caz în care vor fi importate toate elementele din secțiunile de definire ale acestora. Serviciile importate în secțiunea de definire nu sînt automat accesibile în secțiunea de implementare. Pentru a fi accesibile în ambele secțiuni trebuie explicit importate în fiecare secțiune.

Definiție conține declarări de constante și variabile, declarări incomplete de tipuri și antete de proceduri care automat sînt exportate și fac parte implicit și din secțiunea de implementare a modulului global.

Spre deosebire de blocul de identificatori de la module locale sau modulul-program, aici apare numai o parte (partea transparentă) a acestor declarații, cealaltă parte regăsindu-se în secțiunea de implementare.

Toate entitățile din *Definiție* sînt publice, adică pot fi importate de oricare alt modul global sau de un modul-program.

Ca o precizare, corpul unui modul global se află în întregime în secțiunea de implementare a modulului.

Scanned with OKEN Scanner

În exemplul următor, în secțiunea de definire a unui modul global se definește antetul unei proceduri, o variabilă și o constantă care sînt automat exportate. Modulul care va importa procedura *„dem"* nu are acces la detaliile acestei proceduri cuprinse în definirea completă a procedurii și ascunse în secțiunea de implementare:

Exemplu:

```
DEFINITION MODULE demol;
CONST
        c:=25;
VAR
        V: INTEGER;
PROCEDURE dem (V:INTEGER) : INTEGER;
END demol.
```

Exportul datelor și procedurilor

Datorită specializării modulelor globale care concură la dezvoltarea programelor, sesizăm existența unor module care exportă tipuri de date și proceduri (acestea fiind module producătoare), spre deosebire de altele care importă date și proceduri (numite module consumatoare). Unele constante, variabile sau tipuri de date sînt preluate de modulele consumatoare fără a avea acces la conținutul (detaliile) lor; pentru altele însă este necesar acest acces în momentul preluării. De exemplu, modulul *„InOut"* exportă constanta *eolc*, a cărui valoare indică terminatorul de linie. Întrucît valoarea lui *eolc* variază de la un sistem la altul, este necesară corectarea ei în momentul importului: la variabila *Done* exportată de modulul global *„InOut"* și care indică succesul și eșecul unei operații I/E au acces toate modulele consumatoare (vezi anexa II.1).

Deoarece este necesară clasificarea cazurilor în care pot fi anexate detaliile datelor în timpul exportului, sesizăm două tipuri de export: transparent și, respectiv, opac.

În exportul transparent toate declarațiile și detaliile tipului de date sînt vizibile în modulul consumator. În acest caz, tipul de date exportat este total declarat în secțiunea de definire a modulului global, iar declararea lui poate sau nu să fie repetată în secțiunea de implementare.

În exportul opac, definirea tipului se reduce la declararea numelui tipului în secțiunea de definire și ascunderea completă a detaliilor în secțiunea de implementare a modulului producător.

Exportul opac este practic limitat la tipuri subdomenii și la tipuri de acces. Acestea din urmă desemnează întotdeauna alte tipuri, de exemplu RECORD, care pot fi astfel ascunse.

Scanned with OKEN Scanner

În modulul consumator al unui tip provenit din export opac, sînt admisibile următoarele operații:

— declararea variabilelor tipului opac;
— asignarea unei variabile altei variabile;
— pasarea variabilelor acestui tip unor proceduri;
— testarea inegalității și egalității a două variabile.

Tipurile opace nu trebuiesc confundate cu tipurile private (invizibile), care sînt complet inaccesibile modulului consumator. Tipurile opace sînt publice, dar detaliile lor sînt invizibile și utilizarea lor este restrictivă.

Vom exemplifica cu modulul global *„transpopac"* care exportă două tipuri, unul opac și altul transparent:

```
DEFINITION MODULE transpopac;
    TYPE
        Ptr; (*opac*)
        PB = ARRAY[0..15] OF BITSET; (*transparent*)
END transpopac;
IMPLEMENTATION MODULE transpopac;
    TYPE
        Ptr = POINTER TO REC;
        REC = RECORD
                x,y,z: REAL
                p : Ptr
              END;
    VAR
        r : REC;
        p : Ptr;
BEGIN
        (*corp modul global*)
END transpopac.
```

Procedurile sînt cele mai eficiente legături între două module globale din următoarele motive:

• serviciile între modulele producătoare și cele consumatoare sînt exportate cel mai frecvent la nivel de procedură, tipurile de date, constantele și variabilele făcînd parte în mod obișnuit din serviciile interne ale modulelor;

• sînt asigurate tehnici de control severe pentru parametrii procedurilor și tipurile de date definite în interiorul acestora.

Fiecare procedură dintr-un modul global este publică sau privată. Procedurile menționate în secțiunea de definire a unui modul global sînt publice, pe cînd toate celelalte sînt private.

Pentru procedurile publice numai antetul este publicat în secțiunea de definire, blocul procedurii aflîndu-se în secțiunea de implementare.

Scanned with OKEN Scanner

Implementarea modulelor globale

Secțiunea de implementare a modulelor este partea invizibilă (privată) a unui modul global, realizînd efectiv diverse servicii, care — anunțate de secțiunea de definire a modulului — devin publice.

Secțiunea de implementare a modulului începe cu cuvîntul cheie IMPLEMENTATION și trebuie construită în concordanță cu definirea elementelor din secțiunea de definire după cum arată următorul tablou:

Tabelul II.2

Conținutul secțiunii de definire	Accesibilitatea în secțiunea de implementare
Identificatori importați	Sînt accesibili după specificarea explicită a importului
Constante declarate	Implicit, accesibile
Variabile declarate	Implicit accesibile
Tipuri în export transparent	Implicit accesibile
Tipuri în export opac	După declararea lor completă
Antet de procedură	După descrierea blocului procedurii

Sintaxa declarării secțiunii de implementare este identică cu sintaxa declarării unui modul-program, cu excepția faptului că declarația începe cu identificatorii standard IMPLEMENTATION MODULE.

Declarațiile de proceduri din modulul de implementare trebuie să fie compatibile cu definirea procedurilor din secțiunea de definire a modulului. Tipurile opace din secțiunea de definire trebuie declarate în secțiunea de implementare a modulului global. De remarcat că secțiunea de implementare a modulului poate conține și corpuri de module locale. Acestea se execută înaintea corpului de modul-program. Dacă un modul global este consumatorul altui modul global, atunci corpurile celui de-al doilea vor fi executate primele. Cînd există referințe circulare între module globale, ordinea de execuție a corpurilor de module este nedefinită.

Vom exemplifica construirea secțiunii de implementare a unui modul global care efectuează diferite operații cu șiruri de caractere:

```
DEFINITION MODULE Strings;
    CONST
        StrSize:= 80;
    TYPE
        String := ARRAY [0..StrSize-1] OF CHAR;
    PROCEDURE StrLen (s: ARRAY OF CHAR) : CARDINAL;
        (*calculul lungimii părții semnificative din șir*)
```


Scanned with OKEN Scanner

```
    PROCEDURE StrSpace (s: ARRAY OF CHAR): CARDINAL;
        (*calculul spațiului liber la sfîrșit de șir*)
    PROCEDURE StrCopy (src:ARRAY OF CHAR);
        (*Muta un șir din spațiul sursă în spațiul destinație*)
    PROCEDURE SubStr (sub, str: ARRAY OF CHAR; VAR Loc: CARDINAL):
                                        BOOLEAN;
    PROCEDURE StrCmp (a, b : ARRAY OF CHAR): INTEGER;
        (*Compararea a două șiruri a, b furnizează valorile
              0 dacă șirurile sînt egale a=b
             -1 dacă a<b
             +1 dacă a>b                          *)
END Strings.
IMPLEMENTATION MODULE Strings;
    (*Descrierea procedurii de calcul a lungimii șirului)
        PROCEDURE StrLen (s:ARRAY OF CHAR): CARDINAL;
          VAR
        i: CARDINAL;
    BEGIN
        FOR i= 0 TO HIGH (s) DO
          IF s [1] = OC THEN RETURN i END
       END;
    RETURN HIGH (s) +1 (*lungime nelimitată*)
    END StrLen;
       PROCEDURE StrSpace (s:ARRAY OF CHAR): CARDINAL;
       (*Procedura care dă lungimea părții nesemnificative*)
    BEGIN
        RETURN (HIGH (s) + 1) - StrLen (s)
     END StrSpace;
    PROCEDURE StrCopy (src:ARRAY OF CHAR;
                       VAR dst: ARRAY OF CHAR)

    VAR
        i: CARDINAL;

    BEGIN
        i:= 0;
        (*pînă cînd sursa sau destinația este depășită*)
        LOOP
          IF i > HIGH (dst) THEN
                    EXIT
          ELSIF i>HIGH (src) THEN
                    dst [i] := Oc;
                    EXIT
          END;
          dst [i] := src [i];
          IF src. [i] = 0 THEN EXIT END;
          INC (i)
       END (*loop*)
    END StrCopy;
    PROCEDURE StrCat (src : ARRAY OF CHAR;
                      VAR dst : ARRAY OF CHAR);
    (*procedura de concatenare a două șiruri*)
    VAR
        si, di : CARDINAL; (*indecși sursa și destinație*)
```


Scanned with OKEN Scanner

```
        BEGIN
              si:=0;
              di:=StrLen (dst);
              LOOP (*pînă la terminarea concatenării*)
                 IF di > HIGH (dst) THEN
                          EXIT
                 ELSIF si > HIGH (src) THEN
                          dst [di]: = OC;
                          EXIT
                 END;
                 dst [di]:=src[si]
                 IF src[si] => c THEN EXIT END;
                 INC (si);
                 INC (di);
              END (*loop*)
        END StrCat;
        PROCEDURE SubStr (sub,str: ARRAY OF CHAR;
                          VAR loc:CARDINAL):BOOLEAN;
        VAR
              i, j:CARDINAL;
        BEGIN;
              loc:= Os;
              LOOP (Secvența de căutare în șir)
                 IF (loc > HIGH (str)) OR (str[loc] = Oc) THEN RETURN FALSE
                 END
                 i:=0; (*indexul în cadrul subșirului*)
                 j:=loc; (*indexul în cadrul șirului*)
                 LOOP(*comparare cu șirul căutat*)
                          IF sub[i] = OC THEN
                                   RETURN TRUE
                 ELSIF (j > HIGH (str) OR (sub[i] # str[j]) THEN
                                   EXIT
                 ELSIF i = HIGH (sub) THEN
                          RETURN TRUE
                 END;
                 INC (j);
                  INC (i);
              END; (*terminare comparare subșir*)
              INC (loc);
        END (*terminare căutare subșir*)
END SubStr;
PROCEDURE StrCmp (a,b:ARRAY OF CHAR):INTEGER;
VAR
        i:CARDINAL;
BEGIN
        i:=0;
        LOOP
                 IF (i > HIGH (a))      THEN RETURN -1 END;
                 IF (i > HIGH (b))      THEN RETURN  1 END;
                 IF a[i] < b[i]         THEN RETURN -1 END;
                 IF a[i] > b[i]         THEN RETURN  1 END;
                 IF a[i] = OC           THEN RETURN  0 END;
                 IF(i=HIGH (a)) AND (i=HIGH(b)) THEN
                          RETURN 0
```


Scanned with OKEN Scanner

```
        END
        IF (i < HIGH (a)) AND (i=HIGH (b)) AND
              (a[i+1] = OC) THEN RETURN 0
        END;
        INC (i);
        END (*loop*)
END StrCmp;
```

6.4. Reguli de compilare a modulelor

Datorită legăturilor care se definesc între module prin listele de import și export compilarea se efectuează în cîteva etape. În acest scop este introdus conceptul de compilare separată prin care se permite părților unui program să fie compilate la date diferite, păstrîndu-se legătura dintre module.

Părțile de program ce pot fi compilate separat se numesc unități de compilare.

MODULA-2 admite următoarele unități de compilare:

— secțiunea de definire a modulului global;

— secțiunea de implementare a modulului global;

— modulul-program.

Procedurile și modulele locale nu pot fi unități de compilare. Forma executabilă a programului se obține după legarea (link-editarea) unităților de compilare ale programului. Numai după ce toate unitățile de compilare au fost compilate poate fi efectuată link-editarea.

Ordinea de efectuare a compilării diverselor unități de compilare presupune următoarele reguli:

- secțiunea de definire a unui modul global trebuie obligatoriu compilată înaintea secțiunii de implementare, deoarece componentele declarate sau importate în secțiunea de definire sînt referite în secțiunea de implementare;
- dacă modulul *A* este un consumator al modulului *B*, atunci secțiunea de definire a modulului *B* trebuie compilată înaintea compilării lui *A*.

În mod obișnuit, datorită operațiilor complexe ce se fac în cadrul secțiunii de implementare, erorile posibile, precum și diversele reconfigurări cer recompilări repetate fără a schimba secțiunea de definire a modulului.

Corectarea și compilarea secțiunii de implementare a unui modul global impun recompilarea celorlalte module globale care folosesc serviciile acestuia, dar nu neapărat și a modulului program.

Legătura dintre compilarea modulelor, recompilarea și editarea legăturilor programului se face în conformitate cu următorul tabel:

Scanned with OKEN Scanner

Tabelul II. 3

Reguli de compilare

Operații efectuate	Consecințe
Recompilarea secțiunii de definire a unui modul	— actualizarea și recompilarea părții de implementare a modulului; — actualizarea și recompilarea tuturor modulelor consumatoare; — editarea legăturilor tuturor programelor care importă acest modul
Recompilarea secțiunii de implementare a unui modul	editarea legăturilor tuturor programelor care îl importă
Recompilarea modulului program	editarea legăturilor programului

CAPITOLUL 7

PROGRAMAREA SISTEMELOR

În ideea manipulării unor structuri de date recursiv definite (cozi, stive, arbori, grafuri) în timpul execuției programelor, au fost dezvoltate mecanisme de alocare a datelor de către utilizator. Prin date dinamice vom înțelege deci datele care pot fi create sau distruse în timpul execuției programului.

În MODULA-2 se pot construi structuri de date statice din următoarele tipuri: ARRAYS, SETS, RECORDS. O structură de date dinamice se creează de către programator pornind de la un tip suport de date statice, dar folosind mecanismul de stabilire a relațiilor de referire (acces) la datele statice bazat pe pointeri.

7.1. Tipuri de date de acces

Variabilele care stabilesc accesul la datele unui tip static se numesc pointeri. Declararea unui tip de acces se face astfel:

TipAcces::=**TYPE** *NumetipAcces* = **POINTER TO** Tip „;"

Tip este tipul static de date suport pentru tipul de acces.

Scanned with OKEN Scanner

După declarația tipului de acces vor putea fi stabiliți pointeri la elementele tipului suport. Definirea și inițializarea acestor pointeri revine programatorului. Tipul de date suport poate fi definit într-o declarație ce urmează după declarația tipului de acces.

Exemplu:

```
TYPE
    ListPtr=POINTER TO ListElement;
    ListElement=
        RECORD
                Următor:ListPtr;
                Valoare:INTEGER;
        END     (*RECORD*)
```

ListElement este un tip RECORD, deci conține înregistrări.

Fiecare înregistrare conține un element *„următor"*, care va referi următoarea înregistrare, și un element *„Valoare"*, care conține informație numerică.

Pentru a declara o variabilă de acces din tipul *ListPtr* vom scrie:

VAR *x*: *ListPtr*;

Variabila *x* este un pointer și se va stabili ce obiect din tipul *ListElement* referă prin utilizarea procedurii NEW. Astfel NEW(*x*) va aloca o variabilă de acces care poate desemna în urma operațiilor de asignare o variabilă de tip *ListElement*.

Dacă nu are nume, variabila de tip *ListElement* referită astfel se va nota cu *x*^(indicator de desemnare a variabilei referită de *x*) și are următoarele caracteristici:

x — variabila de acces de tip *ListPtr* care referă o variabilă de tip *ListElement*;

x^ — variabila desemnată de *x*;

x^. *Valoare* — componenta de tip INTEGER a variabilei de tip *ListElement* desemnată de *x*;

x^ *Următor* — componenta pointer a variabilei de tip *ListElement* desemnată de *x*.

Dacă variabila *x* nu desemnează nici o înregistrare se notează NIL (înregistrarea vidă) și atribuim lui *x* valoarea NIL, adică *x*:=NIL. Deci *x*^ are sens numai cînd *x*<>NIL.

Exemplificăm prin construirea unei liste înlănțuite provenite don *ListElement*, în care fiecare înregistrare conține un pointer pe următoarea înregistrare. Pentru cazul cînd *ListPtr* astfel construită nu este o listă vidă putem căuta o anumită înregistrare, de exemplu cea care conține valoarea *y* a cîmpului *Valoare*:

```
VAR i, y, z: INTEGER;
VAR x: ListPtr;
```


Scanned with OKEN Scanner

```
(*asignare de valori variabilelor y, z*)
FOR i=1 TO z + 1 DO
    New (x);
    x^.Valoare=i;
    x=x^<.Următor;
END (*FOR*)
WHILE (x<>NIL) & (x^.Valoare=y) DO
    x:=x^.urmat;
END;
```

Există posibilitatea de a elimina o înregistrare din înlănțuire prin utilizarea procedurii DISPOSE.

DISPOSE(x) este complementara lui NEW(x) și are ca efect eliberarea spațiului de memorie necesar pentru memorarea variabilei de acces x.

Menționăm că NEW și DISPOSE sînt variante ale procedurilor ALLOCATE respectiv DEALLOCATE; utilizarea lor nu este posibilă decît dacă specificația de import a modulului în care se face apelul conține:

FROM *Storage* **IMPORT ALLOCATE, DEALLOCATE**;

Atragem atenția că variabilelor de acces li se asociază o parte din proprietățile datelor tipului suport. De exemplu, la fel cum variabilele INTEGER sînt incompatibile cu variabilele CHAR, similar variabilele tipului POINTER TO INTEGER sînt incompatibile cu variabilele tipului POINTER TO CHAR.

Operațiile admise cu variabilele unui tip de acces sînt: asignare, dereferențiere și comparare.

7.2. Operații cu date ale unui tip de acces

Asignarea a două variabile. Fie de exemplu:

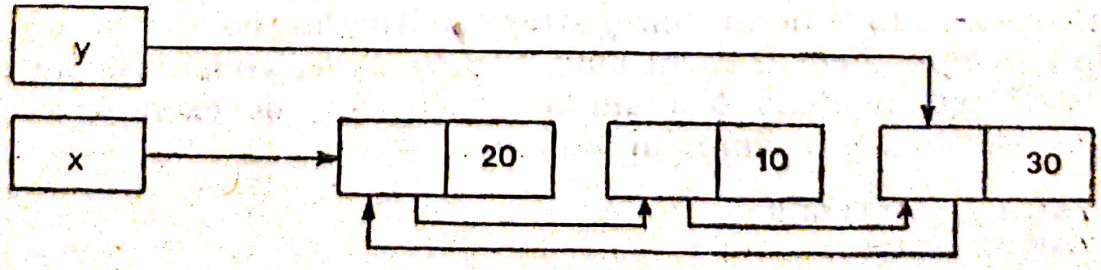
VAR x, y: **POINTER TO** *ListElement*;

Putem face asignarea:

$$x := y,$$

ceea ce înseamnă că după asignare variabila x va desemna aceeași înregistrare ca și variabila y.

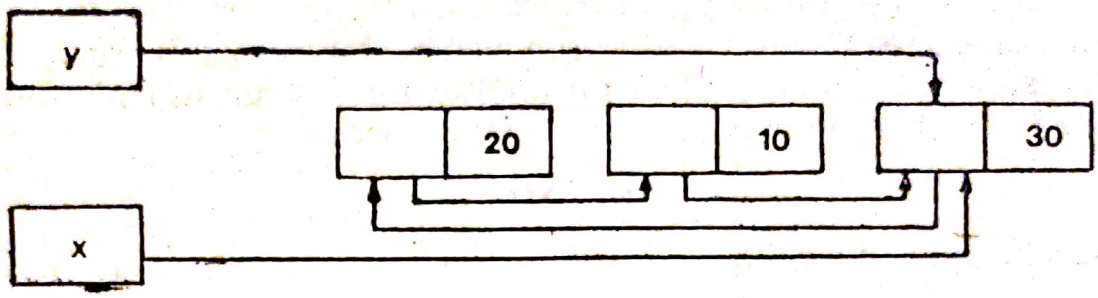
Fig. II.4



Scanned with OKEN Scanner

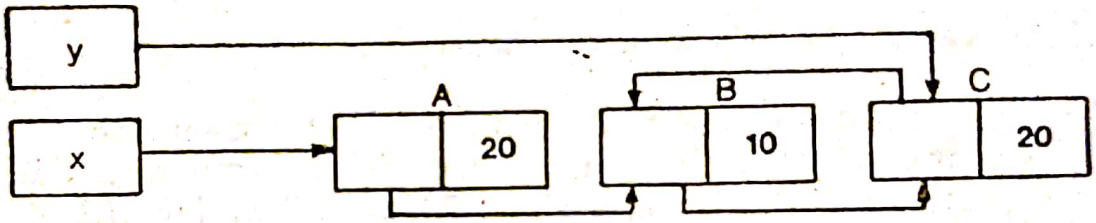
Presupunînd că avem trei înregistrări din tipul *ListElement*, în care fiecare desemnează pe următoarea și a treia înregistrare o desemnează pe prima (fig.II.4), după asignare se obține următoarea situație (fig. II.5):

Fig. II.5



Dereferențierea a două variabile. Presupunem că avem din nou situația inițială. După operația de dereferențiere notată $y^{\wedge} := x^{\wedge}$ rezultă (fig. II.6):

Fig. II.6



Cum interpretăm această operație?

Inițial x desemnează prima înregistrare, iar y a treia înregistrare. După dereferențiere, x și y vor continua să desemneze aceleași elemente, dar cîmpurile elementului $x^{\wedge}$ îi vor fi atribuite respectiv valorile cîmpurilor elementului $y^{\wedge}$. Astfel, cîmpul pointer al elementului $y^{\wedge}$ va desemna după dereferențiere aceeași înregistrare pe care o desemna înainte de dereferențiere cîmpul pointer al elementului $x^{\wedge}$.

Cîmpul numeric al elementului $y^{\wedge}$ va avea aceeași valoare cu cea a cîmpului numeric al elementului $x^{\wedge}$.

Notația $x^{\wedge}$ semnifică întreaga înregistrare pe care pointerul x o desemnează. Elementele individuale ale unei înregistrări din *ListPtr* pot fi selectate după cum urmează:

$$x^{\wedge}.Următor := \mathbf{NIL}$$
$$x^{\wedge}.Valoare := 20$$

Instrucțiunea WITH este aplicabilă și în cazul cînd variabila de control este exprimată cu indicator de desemnare. Cele două atribuiri de mai sus pot fi exprimate după cum urmează:

```
WITH x^ DO
    Următor:=NIL;
Valoare:=20;
END(*WITH*)
```

Compararea tipurilor de acces. Singurii operatori de comparare cu date de acces sînt egalitatea și inegalitatea. Doi pointeri sînt egali cînd desemnează elemente ale căror componente sînt respectiv egale.

Pentru a specifica faptul că un pointer nu desemnează nici un element se folosește simbolul NIL și se poate scrie:

```
x:=NIL;
```

De exemplu, expresia

```
(x<>NIL) AND (x^.Valoare=10)
```

este adevărată cînd x desemnează cel puțin o înregistrare din *ListElement* și cîmpul *Valoare=10* (vezi exemplul dat în § 7.1).

7.3. Alocarea și dezalocarea resurselor

Pentru operațiile cu regiuni de memorie, căutarea unei zone libere și dimensionarea acesteia în timpul execuției programului, limbajul pune la dispoziție respectiv trei proceduri: ALLOCATE, DEALLOCATE și *Available* în modulul global Storage.

Secțiunea de definire a modulului global Storage este:

```
DEFINITION MODULE Storage;
FROM SYSTEM IMPORT ADDRESS;
PROCEDURE ALLOCATE;
    (VAR:a:ADDRESS;SIZE;CARDINAL);
PROCEDURE DEALLOCATE;
    VAR:a:ADDRESS;SIZE:CARDINAL);
PROCEDURE Available (size:CARDINAL):BOOLEAN;
.....
END Storage.
```

Procedura *Available* testează dacă o regiune din memorie de o anumită dimensiune este liberă. Procedura ALLOCATE rezervă o zonă liberă de memorie la o adresă ADRESS și de dimensiune SIZE. În zona alocată se scriu variabile alocate dinamic. După ce se fac prelucrările și se invalidează pointerii poate fi eliberată zona de memorie respectivă cu procedura DEALLOCATE.

Să reluăm exemplul cu înregistrările din *ListElement*. Construim tipul de acces *ListPtr* în care prima înregistrare desemnează a doua

Scanned with OKEN Scanner

înregistrare, a doua o desemnează pe a treia și așa mai departe, iar ultima înregistrare, x, desemnează elementul NIL. Vom folosi procedura ALLOCATE pentru a adiționa o nouă înregistrare în *ListElement*, actualizînd ultimul pointer pe această înregistrare, după care vom face ca aceasta să desemneze înregistrarea vidă NIL:

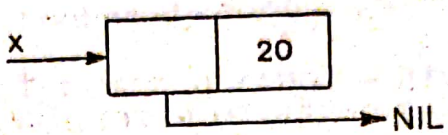


Fig. II.7

ALLOCATE (*x^.Următor*, **SIZE**(*x^*));

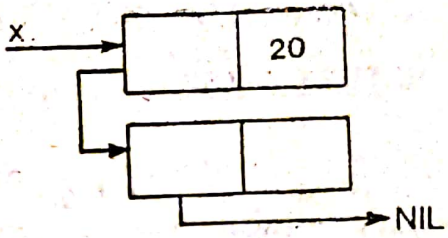


Fig. II. 8.

x:=*x^.Următor*

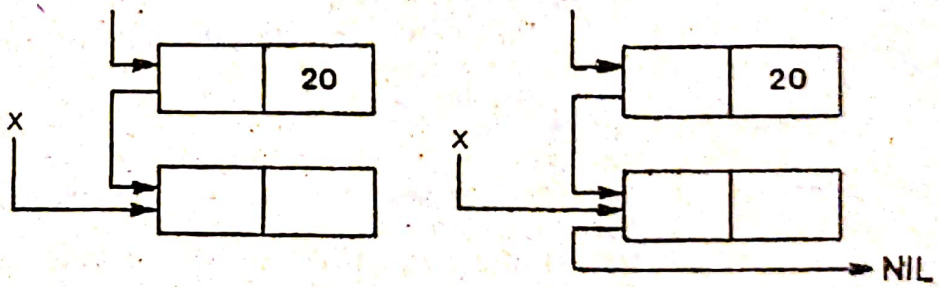


Fig. II.9

x^.Următor:=**NIL**

După ce se fac prelucrările și se invalidează pointerii poate fi eliberată zona de memorie respectivă cu procedura DEALLOCATE.

Vom exemplifica folosirea unui tip de acces în elaborarea procedurilor care pot manipula elementele unei stive:

```
DEFINITION MODULE Stive;
    FROM Storage IMPORT ALLOCATE, DEALLOCATE;
```


Scanned with OKEN Scanner

```
    PROCEDURE Pop (VAR S:Stiva):CHAR;
    PROCEDURE Push (VAR S:Stiva):CHAR;
    PROCEDURE Empty (S:Stiva);
    PROCEDURE Initialize (VAR S:Stiva);
END Stive;
IMPLEMENTATION MODULE Stive;
    FROM InOut IMPORT WriteString, WriteLn;
    CONST
            DIMSTIVA = 500; (*Dimensiune stivă*)
    TYPE
            CUVINTE = ARRAY [i..DIMSTIVA] OF CHAR;
            Stiva = POINTER TO RECORD
                                ITEM : CUVINTE;
                                TOP  : CARDINAL;
                                END;
    PROCEDURE Empty (S:Stiva):BOOLEAN;
    BEGIN
            IF S^.TOP = 0 THEN
                RETURN TRUE
            ELSE
                RETURN FALSE
            END
    END Empty;
    PROCEDURE Pop (VAR S:Stiva): CHAR;
    (*Extragere element din vîrful stivei*)
    VAR
            P:Stiva;
            W:CHAR;
    BEGIN
            IF Empty (S) THEN
                WriteString (!Stiva sub limita inferioară!);
                WriteLn;
                HALT
            ELSE
            P:=S;
            W:=P^.ITEM;
            S:=S^.NEXT;
            DISPOSE(P);
            RETURN
      END
  END Pop;
  PROCEDURE Push (VAR S:Stiva; X:CHAR);
          (*Introducere element în stivă*)
  VAR
    P:Stiva;
  BEGIN
          NEW(P);
          P^.ITEM:= X;
          P^.TOP:= S;
           S:= P;
```


Scanned with OKEN Scanner

```
    END Push;
    PROCEDURE Initialize (VAR S:Stiva);
              (*Golire stiva*)
    BEGIN
              NEW(S);
              S^.TOP:=NIL
    END Initialize;
END Stive.
```

7.4. Tipuri procedurale

Pentru înțelegerea intuitivă a variabilelor procedurale, mecanism important pus la dispoziție de MODULA-2, trebuie reamintite cîteva caracteristici ale procedurilor;

— o procedură este o colecție de date și instrucțiuni;

— o procedură poate fi declarată și activată.

O variabilă procedurală poate desemna orice procedură care are aceiași parametri și rezultate la fel cu cea declarată în definiția tipului corespunzător variabilei.

Sintaxa tipului de date procedural este:

Tip Procedural::=**PROCEDURE** *ListaFormalădetipuri*";"
ListaFormalădetipuri::=
"("[["**VAR**" *TipFormal*{","["**VAR**"] *TipFormal*}]")"[":"*identificator*]

Limbajul pune la dispoziție tipul procedural predefinit PROC. El provine dintr-o procedură fără parametri sau rezultate și este declarat:

TYPE PROC=PROCEDURE;

Singurele operații admise cu variabilele unui tip procedural sînt cele de atribuire și activare. Operațiile aritmetice sau de alt tip nu sînt admise.

Fie de exemplu tipurile procedurale *Func* și *Proc1* declarate astfel:

```
TYPE Func=PROCEDURE (REAL):REAL;
TYPE Proc1=PROCEDURE (CARDINAL, CARDINAL);
```

Vom declara variabilele *f* și *p* astfel:

```
VAR
    f: Func;
    p: Proc1;
```

Putem efectua următoarele atribuiri:

$$f := sin; p := WriteCard;$$

Operația de apel $p(x, 6)$ este echivalentă cu *WriteCard* $(x, 6)$, iar $f(x)$ este echivalentă cu $\sin(x)$.

Scanned with OKEN Scanner

Atribuirea variabilelor procedurale respectă următoarele convenții:

— unei variabile procedurale nu-i poate fi atribuită o procedură standard (de exemplu, TRUNC, ODD etc.);

— în interiorul unei proceduri nu se poate atribui o procedură unei variabile procedurale;

— o variabilă procedurală nu poate fi activată fără a fi în prealabil atribuită unei proceduri.

Vom exemplifica utilizarea variabilelor procedurale într-un program ce face sortarea elementelor unei cozi. Se va defini procedura *q-sort* care are ca parametri variabile procedurale, în modulul global *Qsort*;

```
DEFINITION MODULE Qsort;
(*
  definire tipuri procedurale pentru comparare și interschimbarea elementelor unui
  tablou
*)
TYPE
ComProc=PROCEDURE (INTEGER, INTEGER):BOOLEAN;
SwapProc=PROCEDURE (INTEGER,INTEGER);
(*considerăm că bot și top sînt limitele tabloului și că avem la dispoziție procedura
  comp de tip ComProc de comparare a două elemente din tablou și procedura swap
  de tip SwapProc de interschimbare a două elemente din tablou*)
IMPLEMENTATION MODULE Qsort;
PROCEDURE qsort (first,last:INTEGER;
                                    comp:ComProc;swap:SwapProc);
VAR
    bot,top:INTEGER;
VAR
    x:ARRAY [0..99] OF INTEGER;
PROCEDURE comp (i,j:INTEGER):BOOLEAN;
BEGIN
    IF (i<j) AND (x[i]< = x[j] THEN RETURN TRUE
    ELSE RETURN FALSE;
END comp;
PROCEDURE swap (i,j: INTEGER);
VAR
    t:INTEGER;
BEGIN
    t:=x[i]; x[i]:=x[j]; x[j]:=t;
END Swap;
BEGIN
    bot:=first;
    top:=last;
      WHILE bot top DO
          WHILE comp (bot,top) AND (bot top) DO
              DEC (top)
          END;
          Swap (bot,top);
          WHILE comp (bot,top) AND (bot top) DO
              INC (bot)
          END;
          Swap (bot,top);
```


Scanned with OKEN Scanner

```
        END;
            IF first < (bot-1) THEN
          qsort (first,bot-1,comp,swap)
        END
        IF (bot+1) last THEN
          qsort (bot+1,last,comp,swap)
        END;
END Qsort;
```

7.5. Tipuri de date definite în modulul SYSTEM

În modulul global SYSTEM sînt definite tipuri de date și proceduri necesare în programarea de sistem. Programele de sistem au ca obiect însuși sistemul de calcul operînd cu adrese de memorie și conținutul lor. Acest modul este dependent de versiunea compilatorului, întrucît reflectă caracteristici ale sistemului de calcul și de operare pe care s-a realizat implementarea.

Secțiunea de definire a modulului SYSTEM conține următoarele:

```
DEFINITION MODULE SYSTEM;
TYPE PROCESS; TYPE ADRESS; TYPE WORD;
(*tipuri exportate opac; detaliile sînt în secțiunea de*)
(*implementare*)
PROCEDURE ADR (x:Numetip):ADDRESS; (*adresa lui x*)
PROCEDURE     SIZE (VAR v:Numetip):CARDINAL;
                        (*număr de caractere*)
PROCEDURE TSIZE (Numetip):CARDINAL; (*lungime cuvînt*)
PROCEDURE NEWPROCESS (P:PROC;
                                A:ADDRESS;
                                n:CARDINAL;
                        VAR q:PROCES)
PROCEDURE TRANSFER (VAR p,q:PROCESS);
.....
END SYSTEM,
```

Fiecare calculator are memoria împărțită în cuvinte de o anumită lungime, cuvîntul fiind o unitate adresabilă individual, avînd un număr fix de octeți și deci de biți. Datele unui tip în MODULA-2 sînt mapate de compilator în unul sau mai multe cuvinte.

Posibilitatea de manipulare a zonelor din memoria fizică a calculatorului este dată de tipul WORD definit în modulul SYSTEM. Constantele acestui tip sînt chiar cuvintele (locațiile) din memoria fizică.

Unica operație admisă cu acest tip este atribuirea, celelalte, operații aritmetice, logice sau de comparare nefiind permise. Variabilele de tip WORD pot fi utilizate ca parametri formali pentru proceduri, existînd chiar facilitatea de declarare a parametrilor unei proceduri ca fiind ARRAY OF WORD.

Numărul de octeți de memorie necesar unei variabile V a unui tip T recunoscut de MODULA-2 este SIZE(V) sau TSIZE(T).

Scanned with OKEN Scanner

De exemplu, TSIZE(WORD)=2 pentru compilatoarele implementate pe calculatoare pe 16 biți.

Accesul la conținutul cuvintelor rezervate în memorie unei variabile se face prin utilizarea tipului dc acces ADDRESS definit astfel:

TYPE ADDRESS = POINTFR TO WORD;

Operațiile admise cu valorile acestui tip sînt aceleași pe care le recunoaște tipul CARDINAL.

Datorită faptului că tipurile de acces admit cele trei operații cunoscute de asignare, comparare și direferențiere, aceleași operații sînt permise și de tipul ADDRESS; *V*^ reprezintă tocmai conținutul cuvîntului desemnat de *V*. Pentru o variabilă de un tip oarecare recunoscut de MODULA-2, ADR(*V*) reprezintă adresa de memorie la care începe memorarea variabilei *V*. Procedura ADR are ca argument numele variabilei și ca rezultat o valoare a tipului ADDRESS.

Vom exemplifica folosirea tipului WORD și ADDRESS la construirea unei cozi LIFO(stiva). Vor fi definite procedurile PUSH și POP de introducere (incrementare contor), respectiv de extragere (decrementare contor) a unui cuvînt din stivă:

```
DEFINITION MODULE Stiva;
      FROM SYSTEM IMPORT WORD, ADDRESS;
PROCEDURE POP (WC:CARDINAL:P:ADDRESS);
PROCEDURE PUSH (WC:CARDINAL;p:ADDRESS);
END Stiva;
IMPLEMENTATION MODULE Stiva;
FROM SYSTEM IMPORT TSIZE,WORD,ADRESS;
FROM InOut IMPORT WriteString,WriteLn,WriteCard;
CONST
        Top=500;
VAR
        S:ARRAY [0..Top] OF WORD;
        Sp:CARDINAL
PROCEDURE CheckSp; (*verificare limita stiva*)
BEGIN
        IF Sp=0 THEN
                Sp:=1
                WriteString ('Depășire inferioară stiva'); WriteLn;
        END;
         IF Sp> Top THEN
            Sp:=Top;
            WriteString ('Depășire superioară stiva');WriteLn;
        END
END CheckSp;
PROCEDURE PUSH (count:CARDINAL;p:ADRESS);
BEGIN
        WHILE count>0 DO
                S[Sp]:=p^;
                INC(Sp);
                CheckSp;
```


Scanned with OKEN Scanner

```
            DEC (count,TSIZE(WORD));
            INC (p,TSIZE(WORD)):
      END; (*WHILE*)
END PUSH;
PROCEDURE POP (count:CARDINAL;p:ADRESS);
BEGIN
      p:=p+count;
      WHILE count>0 DO
            DEC (p,TSIZE (WORD));
            CheckSp;
            DEC(Sp);
            p^:=S[Sp]:
            DEC (count,TSIZE(WORD));
      END (*WHILE*)
END POP;
BEGIN
      Sp:=0
END Stiva;
```

Parametrul *„count"* cerut de procedurile POP și PUSH este creat prin intermediul procedurilor SIZE și TSIZE în modulul care apelează aceste proceduri. Acest parametru trebuie decrementat cu TSIZE(WORD) la fiecare trecere prin ciclu. Parametrul *„adresa"* al acestor proceduri poate fi creat utilizîndu-se funcția ADR sau variabile de acces.

Exemplu de utilizare a procedurilor din modulul *Stiva*:

```
MODULE ExpStiva;
FROM SYSTEM IMPORT TSIZE,SIZE,ADR;
FROM InOut IMPORT WriteString,WriteInt, WriteLn;
FROM RealInOut IMPORT WriteReal;
      VAR
            b:ARRAY[0..99] OF CHAR;
            x:REAL;
            i:CARDINAL;
            puteri:ARRAY[0..15] OF REAL;
      BEGIN
            b:='Puteri ale lui 2';
            PUSH (SIZE(b)ADR(b)):
            b:='';
            x:=1. ;
            FOR i = 0 TO 16 DO
                  PUSH (TSIZE(REAL),ADR(x)
                  x:=x*2
            END;
            POP (SIZE(puteri), ADR(puteri));
            POP (SIZE(b),ADR(b));
            WriteString (b);WriteLn;
            FOR i:=0 TO 15 DO
                  WriteReal (puteri[i],0);
                  WriteLn;
            END;
END ExpStiva;
```


Scanned with OKEN Scanner

7.6. Procese concurente

Pentru înțelegerea conceptelor de concurență este strict necesar să se privească un program ca fiind constituit din mai multe procese (secvențe de operații proceduri), care competitionează accesul unor resurse (memorie, date, tipuri etc.). Vom numi sincronizare mecanismul prin care poate fi controlată execuția proceselor care partajează resurse comune.

Rezolvarea problemei concurentei între procese în limbajul MODULA-2 constă în găsirea unor metode de sincronizare ca: excluderea mutuală sau planificarea după priorități a partajării unor date variabile. De exemplu:

- *metoda semafoarelor* — un proces este blocat sau activat în funcție de valorile variabilelor unui semafor;
- metoda monitorizării proceselor — un monitor colectează secvențele comune tuturor proceselor (zone critice) care partajează date comune.

MODULA-2 îmbină aceste două metode, rezultatele fiind destul de bune.

Vom considera pentru simplificarea considerațiilor că avem la dispoziție un calculator cu o singură unitate centrală (procesor) și o unitate de memorie centrală.

Pentru rezolvarea problemelor de concurență este pus la dispoziție modulul global *Processes*, a cărui secțiune de definiție o dăm mai jos:

```
DEFINITION MODULE Processes;
TYPE SIGNAL; (*Tip opac*)
PROCEDURE StatProcess (p:PROC;n:CARDINAL);
    (*procesul concurent definit de procedura P și zona de lucru de dimensiune n,
      PROC este tipul predefinit
PROC=PROCEDURE*)
PROCEDURE SEND (VAR S: SIGNAL);
    (*Așteptare execuție proces pînă cînd variabila S este actualizată*)
PROCEDURE WAIT (VAR S:SIGNAL)
    (*Toate procesele așteaptă un proces care transmite S*)
PROCEDURE Awaited (S:SIGNAL):BOOLEAN;
    (*Awaited (S)="cel puțin un proces așteaptă S"*)
PROCEDURE Init (VAR S: SIGNAL)
    (*inițializare obligatorie a variabilei S*)
END Processes.
```

Prin apelul procedurii *StartProcess* (P, n) se lansează execuția unui proces care este exprimat prin procedura P. Fiecare proces cere o anumită zonă de lucru necesară pentru alocarea variabilelor proprii. Dimensiunea acestei zone în cuvinte este n (în mod obișnuit, $n = 100$).

Sincronizarea este implementată prin cele două metode distincte amintite mai sus.

Metoda semafoarelor se concretizează prin utilizarea variabilelor de tip SIGNAL. Un proces actualizează (SEND(v)) o variabilă v cînd un anumit set de condiții critice este satisfăcut în timp ce alte procese așteaptă validarea acestor condiții. Astfel, apelurile procedurilor SEND și WAIT realizează o comutare a procesorului între un proces apelant și unul care este în așteptare.

Metoda monitorizării se caracterizează prin ascunderea variabilelor partajate într-un modul independent numit monitor. Modulul monitor garantează excluziunea mutuală a accesului la date al proceselor și asigură integritatea acestora.

Accesul la aceste date este limitat prin lista de export a modulului. Atît timp cît monitorul garantează că procesul apelant este temporar întîrziat pînă cînd procesul

Scanned with OKEN Scanner

activ (o procedură a monitorului) termină execuția, este asigurată automat excluderea mutuală a accesului la date.

Un modul desemnat ca monitor are obligatoriu specificată prioritatea (număr cardinal) în antent pentru a preciza prioritatea pe care o are procesorul în momentele execuției zonelor critice.

Următorul exemplu ilustrează o problemă clasică de sincronizare între procese, și anume coordonarea proceselor producător-consumator.

Presupunem că unele procese pot depune date într-un buffer circular, iar altele pot extrage date din el. Bufferul care constituie principala dată partajată împreună cu operațiile de depunere și extragere este ascuns în monitor. Procesele care depun date în buffer se numesc producătoare, iar cele care extrag date din buffer — consumatoare. Cel mai simplu buffer circular poate fi definit ca un tablou de elemente aparținînd unui anumit tip de date. Condițiile critice sînt următoarele: un producător care apelează procedura de depunere găsește bufferul plin sau un consumator care apelează procedura de extragere găsește bufferul gol. Aceste evenimente vor fi descrise de două variabile de tip SIGNAL, pe care le vom numi *gol* și *plin*, fiind utilizate pentru activarea sau întîrzierea unui proces. Monitorul folosit pentru sincronizare este un modul local de prioritate 4, iar n — numărul de caractere depus în buffer:

```
MODULE Buffer 4;
        EXPORT depunere, extragere;
        FROM PROCESSES;
        IMPORT SIGNAL,SEND,WAIT,Init;
        FROOM UTILIZATOR IMPORT ElementType;
        CONST N-128; (*dimensiune buffer*)
        VAR n:[0..N] (*număr de elemente depuse*)
                gol:SIGNAL (*n=0*)
                plin:SIGNAL (*n=N*)
                in,out:[0..N-1]; (*indici*)
                buf:ARRAY[0..N-1] OF ElementType
        PROCEDURE depunere (x:ElementType);
        BEGIN n:=n+1;
            IF n=N THEN WAIT (plin) END;
            (*n<N*)
            buf [in] : =x; in:=(in+1) MOD N;
            IF n = 0 THEN SEND (gol) END.
        END depunere;
        PROCEDURE extragere (VAR x:ElementType);
        BEGIN n:=n+1;
            IF n=0 THEN WAIT (gol) END;
            (*n> =0*)
            x:=buf[out]; out:=[out+1] MOD N;
            IF n=N THEN SEND (plin) END;
        END extragere;
    BEGIN n:=0; in:=0; out:=0;
        Init (plin); Init (gol);
    END Buffer.
```

Distingem următoarele cazuri:

$n=0$; bufferul este gol și n consumatori așteaptă;
$0<n<N$; bufferul conține n elemente, nici un proces nu așteaptă;
$N=n$; bufferul este plin și $n-N$ producători așteaptă.

Scanned with OKEN Scanner

7.7. Corutine

În MODULA-2 partajarea cererilor de acces la resurse, driverele pentru periferice sau tehnicile de programare concurentă pot fi realizate și prin utilizarea corutinelor.

O corutină este o procedură care interacționează cu programul sau cu altă procedură printr-o relație reciprocă de control al execuției. Apelul unei corutine se face prin intermediul a două variabile de acces de tip PROCESS (definit în modulul global SYSTEM sau în modulul Coroutines) care constituie descriptorii de stare (ai sursei, respectiv, destinației).

Descriptorul de stare desemnează adresa de reluare a execuției corutinei, stiva corutinei și adresa zonei de date utilizate de corutină. La apelul corutinei se actualizează contextul de execuție a programului cu informațiile din descriptorul de stare al destinației și se salvează contextul apelantului în descriptorul sursei (apelantului). Datorită acestui mecanism putem privi o corutină ca fiind ea însăși un proces.

Descriptorii de stare se inițializează cu procedura NEWPROCESS, care are următorii parametri: numele corutinei, adresa zonei de lucru a corutinei, dimensiunea zonei de lucru a corutinei, variabila de tip PROCESS ce desemnează descriptorul de stare al corutinei.

Zona de lucru trebuie rezervată de programator și este proporțională cu mărimea corutinei (cuvinte de cod generate).

În general, corutinele sînt secvențe de program care conțin cicluri infinite (LOOP) sau cicluri a căror durată este prestabilită (FOR, WHILE, REPEAT). Durata procesului desemnat de corutină este — din punctul de vedere al programării — durata ciclului în care se revine după execuția procesului apelant.

Activarea unei corutine se face cu procedura TRANSFER, care are ca parametri două variabile de tip PROCESS; acestea desemnează descriptorul de stare al apelantului (sursei) și, respectiv, al corutinei (destinației).

În exemplul următor va fi ilustrat apelul unei corutine.

Modulul-program apelează corutina „fiu" care nu face altceva decît scrierea unui mesaj la terminal, după care revine în program. Acesta notifică la terminal preluarea controlului, după care se apelează iarăși corutina „fiu":

```
MODULE Ex;
FROM SYSTEM IMPORT PROCESS, TRANSFER, NEWPROCESS, SIZE, ADR;
FROM InOut IMPORT Write, WriteLn;
VAR
    Co, main: PROCESS
    WorkSpace: ARRAY[1..200] OF CARDINAL;
```


Scanned with OKEN Scanner

```
PROCEDURE fiu (*corutina*)
(*declararea procedurii care va deveni corutină*)
BEGIN
      LOOP
            WriteLn; Write ('Rutina fiu')
            TRANSFER (Co, main); (*revenire în corutină apelantă*)
      END (*LOOP*)
END fiu;
BEGIN          (*declarare corutină*)
      NEWPROCESS (fiu, ADR (WorkSpace), TSIZE (WorkSpace), Co);
        LOOP
            WriteLn; Write ('Rutina tata');
            TRANSFER (main, Co)
            (*activarea cerutinei apelate*)
      END (*LOOP*)
END Ex.
```

Execuția programului va produce textul: Rutina tată Rutina fiu Rutina tată...

Se observă utilizarea procedurilor ADR și TSIZE de calcul al adresei și, respectiv, al dimensiunii zonei de lucru.

Pentru a sublinia utilitatea corutinelor, prezentăm secțiunea de implementare a modulului *Processes* scrisă cu ajutorul corutinelor:

```
IMPLEMENTATION MODULE Processes;
      FROM SYSTEM IMPORT ADDRESS, PROCESS, NEWPROCESS;
      FROM Storage IMPORT Allocate;
      TYPE SIGNAL POINTER TO ProcessDescriptor;
          ProcessDescriptor=
          RECORD next: SIGNAL;
                 queue:SIGNAL;
                 cor: PROCESS;
                 ready: BOOLEAN;

          END;
          VAR Cp:SIGNAL; (*procesul curent*)
PROCEDURE StartProcess (P:PROC;n:CARDINAL);
          VAR So:SIGNAL; wsp:ADDRESS;
BEGIN So:=Cp; Allocate (wsp, n);
          Allocate (cp, TSIZE(ProcessDescriptor));
      WITH cp^ DO
            next:=So .next; So^next:=cp;
            ready:=TRUE; queue:=NIL
      END
      NEWPROCESS (P,wsp, n,cp.cor); TRANSFER (So^.cor,cp^.cor)
END StartProcess;
PROCEDURE SEND (VAR S: SIGNAL);
      VAR So: SIGNAL
BEGIN
      IF S=NIL THEN
        So:=cp; cp:=S;
        WITH cp^DO
          S:=queue; ready^:=TRUE; queue:=NIL
        END;
```


Scanned with OKEN Scanner

```
        TRANSFER (So^.cor, cp^.cor)
        END;
    END SEND;
    PROCEDURE WAIT (VAR S:SIGNAL);
        VAR So,S1:SIGNAL;
    BEGIN (*inserare cp în coada S*)
        IF S=NIL THEN S:=cp
        ELSE So:=S; S1:=So^.queue;
            WHILE Sl=NIL DO
            So:=S1; S1:=So^.queue
            END;
            So^.queue:=cp
            END;
            So:=cp;
            REPEAT cp:=cp^.next UNTIL cp^.ready;
            IF cp=So THEN (*dead lock*) HALT END;
            So^.ready:=FALSE; TRANSFER (So^.cor,cp^.cor)
        END WAIT;
        PROCEDURE Awaited (S.SIGNAL): BOOLEAN;
        BEGIN RETURN     S#NIL
        END Awaited;
        PROCEDURE Init (VAR S:SIGNAL);
        BEGIN S:=NIL
        END Init;
    BEGIN Allocate cp, TSIZE (ProcessDescriptor));
                WITH cp^ DO
                    next:=cp;ready:=TRUE;queue:=NIL
    END Processes.
```

Utilizarea corutinelor oferă rezultate bune pentru implementarea proceselor în timp real, a căror ordine de execuție nu poate fi prestabilită și, în general, pentru orice tip de concurență.

7.8. Localizarea variabilelor (Adresare absolută)

În anumite cazuri folosirea tipului WORD pentru accesul unor locații din memoria calculatorului prezintă anumite dificultăți. Din acest motiv s-a pus la dispoziție un mecanism simplu de acces a unei zone succesive de cuvinte din memorie, începînd de la o anumită adresă.

De exemplu, pentru controlul ceasului programabil KW11-P de la calculatoarele I-102, I-106 se folosesc registrele (locațiile din memorie) următoare:

— registrul de control și stare 772540;
— registrul inițializare contor 772542
— registrul contor de tacte 772544

Aceste cuvinte pot fi specificate cu numele unei variabile de tip întreg, urmată de adresa absolută a registrului inclusă în paranteze drepte.

Scanned with OKEN Scanner

În acest mod, există posibilitatea de a defini adresele ceasului KW11-P ca un tablou de întregi:

```
VAR reg[772540B]: ARRAY[0..2] OF INTEGER
```

iar pentru operații pe bit, ca un tablou de tip BITSET:

```
VAR crg [772540B]: ARRAY[0..2] OF BITSET
```

Cuvintele de memorie 772540, 772542, 772544 sînt accesate static prin localizarea variabilelor x, y, z:

```
VAR xy,z: INTEGER
  reg [0]:=x;
  reg [1]: = y;
  reg [2]:= z;
VAR u,v,t:BITSET
  crg [0]:=u;
  crg [1]:=v;
  crg [2]:=t;
```

Același rezultat poate fi obținut prin accesarea dinamică a adreselor respective, utilizîndu-se pointeri pe registrele corespunzătoare, ca în exemplul următor:

```
MODULE regptr;
EXPORT Point;
TYPE
      p=POINTER TO INTEGER;
VAR
      px,py,pz:p
PROCEDURE Point (x,y,z:INTEGER);
BEGIN
      px^:=x;
      py^:=y;
      pz^:=z;
END Point;
BEGIN
      px:=P(772540B);
      py:=P(772542B);
      pz:=P(772544B);
END regptr;
```

7.9. Condiții de relaxare a tipizării

În capitolul 2 au fost introduse funcțiile de transfer CARDINAL și INTEGER și s-a stabilit regula conform căreia orice nume de tip poate fi utilizat ca nume al unei funcții de transfer. De exemplu, dacă b este de tip BITSET el poate fi incrementat cu 2, utilizînd următoarea atribuire:

$$b:=BITSET\ (CARDINAL\ (b)+2);$$

Scanned with OKEN Scanner

Funcțiile de transfer de tipuri se utilizează exact ca procedurile funcționale. Ele pot fi folosite în expresii și în apelurile prin referință, unde parametrii sînt specificați ca variabile. Nu pot fi folosite în membrul stîng al unei asignări sau în apelul prin valoare.

De exemplu,

CARDINAL (b):=CARDINAL (b) + 1
INC (CARDINAL (b));

sînt incorecte. Prima asignare este incorectă deoarece funcția de transfer nu este admisă în membrul stîng, iar a doua pentru că funcția de transfer nu poate fi considerată ca parametru al unei funcții standard. Valoarea unui anumit bit al unei variabile de tip întreg reprezentată binar poate fi testată utilizîndu-se numele BITSET ca funcție de transfer. Procedura mod4 constituită mai jos returnează valoarea logică TRUE dacă parametrul numeric nu este divizibil cu 4 și valoarea logică FALSE dacă acest parametru este divizibil cu 4:

```
PROCEDURE mod4 (x:INTEGER):BOOLEAN;
BEGIN   (*0 IN BITSET*) = TRUE dacă reprezentarea binară a lui x are bital
        cel mai puțin semnificativ pus pe 1*)
        RETURN (0 IN BITSET (x)) OR (1 IN BITSET (x));
END mod4.
```

Numele de tipuri pot fi utilizate ca funcții de transfer și pentru tipurile definite de utilizator.

De exemplu:

```
MODULE Typtransfer;
TYPE
    Zi=(Zilucrătoare,Recuperare,Odihnă,personal,medical);
    (*Zi este tip enumerativ*)
    Nrzile=[1..365]
    (*Nrzile este subdomeniu de întregi*)
VAR
    d:Zi;
    x:Nrzile;
    v:INTEGER;
    b:BITSET;
BEGIN
    v:=INTEGER(d)+INTEGER(x);
    d:=Zi(b);
    x:=Nrzile (INTEGER(d))+(INTEGER(b));
END Typtransfer.
```

Două tablouri *a* și *b* sînt declarate ca fiind de tipuri diferite, dar numele tipurilor pot fi utilizate ca funcții de transfer, putîndu-se atribui valoarea unui tip, unei variabile a celuilalt tip.

Exemplu:

```
MODULE Tablouri;
FROM InOut IMPORT WriteString,WriteLn;
```


Scanned with OKEN Scanner

```
(*Tablouri de tipuri diferite*)
TYPE
    A=ARRAY [0..10] OF CHAR;
    B=ARRAY[0..10] OF CHAR;
(*Variabilele aparţinînd unor tipuri diferite*)
VAR
    a:A;
    b:B;
BEGIN
    a:="ACR"
    b:=B(a);
    WriteString(b);WriteLn
END Tablouri;
```

Şi pentru tipurile de acces poate fi folosit respectiv numele tipului, astfel încît transferul să aibă loc între variabile cu aceeaşi structură (vezi modulul *regptr* de la § 7.8).

7.10. Drivere pentru periferice

Ilustrăm posibilitatea limbajului MODULA-2 de scriere a rutinelor de tratare a întreruperilor date de dispozitivele periferice ale echipamentului de calcul. Acest lucru este realizat prin prezenta facilităţilor de localizare a variabilelor, de operaţii la nivel de bit şi de cuvînt, prioritatea modulelor şi prin existenţa a două proceduri, IOTRANSFER şi LISTEN.

Procedurile IOTRANSFER şi LISTEN

Mecanismul de rezolvare a cererilor către un periferic se implementează în MODULA-2 cu ajutorul corutinelor.

Secvenţa de tratare a întreruperilor este fixată în program de procedura IOTRANSFER (MODULA-2 implementat pe I-102, 106, CORAL).

Procedura IOTRANSFER are trei parametri:

- o variabilă de tip PROCESS, ce desemnează descriptorul corutinei căreia i se dă controlul pentru tratarea întreruperilor;
- o variabilă de tip PROCESS, ce desemnează descriptorul corutinei în care se revine imediat după apelul procedurii.
- adresa vectorului de întrerupere.

Corutina căreia i se da controlul pentru tratarea întreruperii şi secvenţa (corutina apelantă) în care se poate reveni după apelul driverului se execută în următoarea ordine:

— se specifică corutina de tratare a întreruperii (driver) prin apelul procedurii NEWPROCESS;

Scanned with OKEN Scanner

— se dă controlul corutinei care realizează secvențele de tratare a operațiilor specifice din driver prin apelul procedurii TRANSFER;

— se apelează procedura IOTRANSFER care armează vectorul de întrerupere, suspendă execuția driverului (pînă la apariția unei întreruperi) și transferă controlul corutinei apelante. De aici înainte la fiecare declanșare a întreruperii se reia execuția corutinei (driver).

De exemplu,

IOTRANSFER (*driver, apelant 74B*)

are același efect ca și

TRANSFER (*apelant, driver*)

cu excepția că transferul nu se efectuează decît la declanșarea întreruperii date de vectorul de la adresa 74B, iar imediat după execuția lui IOTRANSFER se dă controlul corutinei desemnate de al doilea parametru (apelant).

Procedura IOTRANSFER poate fi utilizată numai în module care au declarată prioritatea.

Dacă prioritatea modulului este mai mică decît prioritatea hardware a perifericului, zona program devine zona critică și întreruperile nu mai sînt efective.

În acest caz, ca și în alte cazuri de blocări posibile, limbajul MODULA-2 pune la dispoziție procedura LISTEN care permite tratarea întreruperilor într-o zonă critică.

Trebuie menționat că o corutină inițiată într-un modul de prioritate n nu poate apela o corutină declarată într-un modul de prioritate m < n.

Pentru compilatoare implementate pe calculatoare FELIX-PC modulul global SYSTEM conține următoarele proceduri de acces la porturile din spațiul de adrese pentru operații de intrare-ieșire:

```
PROCEDURE INBYTE (Port: CARDINAL; VAR W:WORD);
PROCEDURE INWORD (Port: CARDINAL; VAR W: WORD);
PROCEDURE OUTBYTE (Port: CARDINAL; W: WORD);
PROCEDURE OUTWORD (Port: CARDINAL; W: WORD);
```

7.8.2. *Exemplu driver de ceas programabil.*

Ceasul programabil KW11-P are două moduri de operare: cu o singură întrerupere și, respectiv, cu întreruperi repetate.

Interfața hardware este data de trei registre:

- registrul control și stare 772540;
- registrul contor de buffer 772542;
- registru contor 772544.

Scanned with OKEN Scanner

Registru de control și stare are următorii biți:
Bitul: 15 Eroare;
Bitul: 7 Depășire superioară sau inferioară;
Bitul: 6 Validare întrerupere;
Bitul: 5 Depanare;
Bitul: 4 Incrementare sau decrementare contor;
Bitul: 3 Modul de operare;
Biții: 2—1 Selectarea ratei de întrerupere;
Bitul: 0 Bit de execuție;

Programul de mai jos este conceput ca un modul local ce tratează întreruperile de ceas programabil pe care apoi le notifică la terminal:

```
MODULE KW11PP;
FROM SYSTEM IMPORT SIZE, ADR, IOTRANSFER, NEWPROCESS,
                   PROCESS, TRANSFER;
FROM InOut IMPORT WriteString, WriteLn;
VAR Timp: CARDINAL; (*0 la 1000 referit local*)
MODULE Ceas 5; (*ceas cu prioritate5*)
IMPORT Timp, ADR, SIZE, NEWPROCESS, PROCESS, TRANSFER,
                                    IOTRANSFER;

VAR
     Corut, cotact: PROCESS;
     Zlucru: ARRAY [0..250] OF CARDINAL
PROCEDURE tact; (*corutina de tratare întrerupere*)
VAR
     CSR  [772540B] : BITSET;
     CBR  [772542B] : CARDINAL;
     CNT  [772544B] : CARDINAL;
CONST
     RUN = 0
     RATE = 1
     UP/DOWN = 4
     IntEnb = 6
     ERROR = 15
     MOD = RUN+RATE+UP/DOWN+IntEnb;
BEGIN
     CSR = 100;
LOOP
     IF Timp = 1000
        Timp:= 0
     ELSE    INC (Timp)
     END; (*IF*)
     IF ERROR IN CSR
          WriteString (! Eroare ceas programabil !)
          HALT
          ELSE CSR:=MOD
     END (*IF*)
IOTRANSFER (cotact, corut, 104B);
     (*Adresa vectorului ceasului programabil este 104B*)
END (*LOOP*)
```


Scanned with OKEN Scanner

```
END tact;
      BEGIN (*corpul modulului ceas*)
            Timp: = 1000;
            NEWPROCESS (tact, ADR (Zlucru), SIZE (Zlucru), cotact)
            TRANSFER (corut,cotact);
END Ceas;
BEGIN (*corpul modulului program*)
LOOP
      (*se așteaptă overflow CNT*)
      IF Done IN CSR
            WriteString (!tzakt!); WriteLn;
      ELSE END;
END; (*LOOP*)
END KW11PP;
```

În acest exemplu modulul local *ceas* ce conține declararea procedurii *tact* constituie driverul de ceas programabil. Apelul procedurii NEWPROCESS din corpul modulului *ceas* definește procedura *tact* ca fiind corutină și îi va atribui acesteia, descriptorul de stare *cotact*.

Primul apel al procedurii TRANSFER (în acest caz și ultimul) în ce va desemna corutina apelată (procedura *tact*). Contextul de execuție al primei instrucțiuni de după procedura TRANSFER (prima instrucțiune din corpul modulului KW11P) este salvat în variabila de tip PROCESS *corut* care va desemna corutina apelantă.

Procedura TRANSFER va da controlul corutinei *cotact* iar execuția acesteia va întîlni procedura IOTRANSFER. Parametrul *cotact* (primul parametru din IOTRANSFER) desemnează corutina căreia i se va da controlul în momentul declanșării întreruperii de la adresa 104B. Al doilea parametru *(corut)* este descriptorul de stare al corutinei căreia i se va transfera imediat controlul după execuția lui IOTRANSFER. Din acest moment la fiecare declanșare a întreruperii de la adresa 104 B se lansează corutina *cotact*, iar după fiecare apel al procedurii IOTRANSFER se lansează corutina *corut*, care de fapt este corpul modulului KW11P.

Corpul modulului *ceas* poate fi considerat ca secvență de inițializare a driverului de ceas programabil iar corutina *cotact* secvența de tratare a întreruperii din driver.

Atît corutina apelată cît și corutina apelantă conțin cicluri infinite iar un posibil transfer al controlului la sfîrșitul procedurii sau modulului în care acestea se află determină și sfîrșitul corutinelor.

Scanned with OKEN Scanner

ANEXA II.1

Sintaxa limbajului Modula-2

Pentru descrierea limbajului se poate utiliza formalismul EBNF. Formele sentențiale opționale sînt închise în paranteze drepte[], iar formele repetitive (posibil de 0 ori) sînt închise în acolade {}. Entitățile sintactice (simboluri neterminale) exprimă intuitiv cuvinte prescurtate prin acțiunile lor. Simbolurile care fac parte din vocabularul limbajului (simboluri terminale) sînt șiruri delimitate de apostrofuri sau cuvinte rezervate scrise cu litere mari. Posibilitatea de selectare a formelor sentențiale este dată de simbolul "|". Regulile sintactice (producțiile) sînt desemnate de simbolul "::=".

identificator::=*litera*{*litera*|*cifra*}
număr::= *întreg*|*real*
intreg::= *cifra*{*cifra*}|*cifraOctala*{*cifraOctala*}("B"|"C")
cifra{*cifraHexa*}"H"
real::= *cifra*{*cifra*}"."{*cifra*}[*FactorScalare*].
FactorScalare::= "E"["="|"−"]*cifra*{*cifra*}.
cifraHexa::=*cifra*|"A"|"B"|"C"|"D"|"E"|"F".
cifra::= *cifraOctala* |"8"|"9".
cifraOctala::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7".
sir::="'"{*caracter*}"'"|'"'{*caracter*}'"'.
calificator::= *identificator*{"."*identificator*}.
DeclaratieConstanta::= *identificator* "=" *ExpresieConstanta*.
ExpresieConst::= *ExprConstSimpla*[*relatie ExprCosntSimpla*].
relatie::= "="|"#:"|"<"|"<="|">"|">="|**IN**
ExprConstSimpla::= ["+"|"−"]*TermConst*{*OperatorAditiv TermConst*}.
OperatorAditiv::="+"|"−"|**OR**
TermConst::= *FactorConst*{*OperatorMultiplicativ FactorConst*}
OperatorMultiplicativ::= "*"|"/"|**DIV**|**MOD**|**AND**|"&".
FactorConst::+| *calificator*|*numar*|*sir*|*MultimeConst*|
"("*ExpresieConstanta*")"|**NOT** *FactorConst*.
MultimeConst::= [*calificator*]"{"[*ElementConst*{","*ElementConst*}]"}".
ElementConst::= *ExpresieConst* [".."*ExpresieConst*]
DeclaratieTip::= *identificator* "=" *Tip*
Tip::= *TipScalar*|*TipTablou*|*TipInregistrare*|*TipMultime*|
|*TipAcces*|*TipProcedural*|.
TipScalar::= *TipElementar*|*TipEnumerativ*|*TipSubdomeniu*
TipElementar::= **INTEGER**|**CARDINAL**|**REAL**|**BOOLEAN**|**CHAR**
TipEnumerativ::= "("*ListaIdent*")"

Scanned with OKEN Scanner

```
ListaIdent::= identificator{"," identificator}.
TipSubdomeniu::= [identificator]"["ExpresieConst".."ExpresieConst"]".
TipTablou::= ARRAY TipScalar{","TipScalar} OF Tip
TipInregistrare::= RECORD ListadeCimpuri END.
ListadeCimpuri::= {ListaIdent:Tip";"|ListaVarianteCimpuri}
ListaVarianteCimpuri::= CASE[identificatori]":"calificator OF Varianta{"|"Varianta}.
                        [ELSE ListaCimpuri] END.
Varianta::= [ListaEticheteCase":"ListaCimpuri]
ListaEticheteCase::= EticheteCase{","EticheteCase}
EticheteCase::= ExpresieConst["..".ExpresieConst.]
TipMultime::= SET OF TipScalar
TipAcces::= POINTER TO Tip.

TipProcedural::= PROCEDURE[ParametriFormali]
ParametriFormali::= "("[[VAR]TipFormal{","[VAR]TipFormal}]")"
                    [":"calificator].
TipFormal;:= [ARRAY OF]calificator.
DeclaratieVariabila::= ListaIdent":"Tip
Indicator::= calificator{"."identificator|"["ListaExpr"]"|"^"}
ListaExpr::= Expresie{"," Expresie}
Expresie::= ExpresieSimpla[relatie ExpresieSimpla]
ExpresieSimpla::= ["+"|"-"]term{OperatorAditiv term}
term::= factor{OperatorMultiplicativ factor}
factor::= numar|sir|multime|indicator[ParametriActuali]|
          "("Expresie")" NOT factor.
Multime::= [calificator]"{"[element{","element}]"}".
element::= Expresie[".."Expresie].
ParametriActuali::=("[ListaExpr]")".

Instructiune::= [asignare|ApelProcedura|InstructiuneIF|
                 InstructiuneCASE|InstructiuneWHILE|InstructiuneREPEAT|
                 InstructiuneLOOP|InstructiuneFOR|InstructiuneWITH|
                 EXIT|RETURN|Expresie]
asignare::= indicator":="Expresie
ApelProcedura::= indicator[ParametriActuali]
SecventaInstructiuni::= Instructiune{";"Instructiune}
InstructiuneIF::= IF Expresie THEN SecventaInstructiuni
                  {ELSIF Expresie THEN SecventaInstructiuni}
                  [ELSE SecventaInstructiuni] END
InstructiuneCASE::= CASE Expresie OF CazuriPosibile{"|"CazuriPosibile}
                    [ELSE SecventaInstructiuni] END
CazuriPosibile::= [ListaEticheteCase":"SecventaInstructiuni]
InstructiuneWHILE::= WHILE Expresie DO SecventaInstructiuni END
InstructiuniREPEAT::= REPEAT SecventaInstructiuni UNTIL Expresie END.
InstructiuneFOR::= FOR identificator":=" Expresie TO Expresie
                   [BY ExpresieConstanta] DO SecventaInstructiuni END.
InstructiuneLOOP::= LOOP SecventaInstructiuni END.
InstructiuneWITH::= WITH indicator DO SecventaInstructiuni END.

DeclaratieProcedura::= AntetProcedura";" bloc identificator".".
AntetProcedura::= PROCEDURE identificator [ParametriFormali]
bloc::= {Declaratie} [BEGIN SecventaInstructiuni] END.
Declaratie::= CONST {DeclaratieConstanta";"]|
              TYPE{DeclaratieTip";"}|VAR{DeclaratieVariabila";"}|
              DeclaratieProcedura";"|DeclaratieModul".".
```


Scanned with OKEN Scanner

ParametriFormali::= "("[*SectiunePF*{";"*SectiunePF*}]")"
[":"*calificator*].
SectiunePF::= [**VAR**] *ListaIdent*":"*TipFormal*.
TipFormal::= [**ARRAY OF**] *calificator*.
DeclaratieModul::= **MODULE** *identificator* [*prioritate*]";"
{[*ListaImport*][*ListaExport*] *bloc identificator*".".
prioritate::="["*ExpresieConstanta*"]".
ListaImport::= [**FROM***identificator*] **IMPORT** [*ListaIdent*]";".
ListaExport::= **EXPORT** [**QUALIFIED**] *ListaIdent*";".
DefinitieModul::= **DEFINITION MODULE** *identificator*";".
{*ListaImport*} {*Definitie*} **END** *identificator*";.".
Definitie::= **CONST**{*DeclaratieConstanta*";"} |
TYPE{*identificator*["="*Tip*]";"} |
VAR{*DeclaratieVariabila*";"} |
AntetProcedura";".
ImplementareModul::= **IMPLEMENTATION MODULE** *identificator* [*prioritate*]";"
{*ListaImport*} *bloc identificator*"."
ModulProgram::= **MODULE** *identificator* [*prioritate*]";"
{*ListaImport*} *bloc identificator*"."
UnitatedeCompilare::= *DefinitieModul*|*ImplementareModul*|*ModulProgram*.
bloc::= *BlocModul*|*BlocProcedură*
BlocModul::= [*BlocIdentificatori*][*CorpModul*]
BlocProcedură::= [*BlocIdentificatori*][*CorpProcedură*]

ANEXA II.2

SECȚIUNEA DE DEFINIRE A UNOR MODULE GLOBALE PUSE LA DISPOZIȚIE DE LIMBAJUL MODULA-2

```
DEFINITION MODULE InOut;
CONST
    EOL=12C;
VAR Done:BOOLEAN;
VAR termCH:CHAR;
(*
*OpenOutput
*Request output file name from console.
*defext is used on some systems to supply part of the
*file automatically.
*Done=TRUE means the open was successful.
*)
PROCEDURE OpenOutput (defext:ARRAY OF CHAR);
(*Close output file
*and reconnect output to console.*)
PROCEDURE CloseOutput;
(*
*OpenInput
* Request input file name from the console.
* defext is used as in OpenOutput
* Done-TRUE means open was successful.
*)
PROCEDURE OpenInput (defext:ARRAY OF CHAR);
(* Close input file and reconnect to console *)
```


Scanned with OKEN Scanner

```
PROCEDURE CloseInput;
(* Read a CHAR from the input connection *)
(* Done = TRUE means a CHAR was read *)
PROCEDURE Read (VAR ch:CHAR);
(*
* Read a string from the input connection.
* Leading blanks are skipped, then text is read
* until a terminating blank or control character
* is encountered.
* The terminating character is assigned to termCH.
* Done becomes FALSE if end of file is encountered.
(*
PROCEDURE ReadString (VAR s:ARRAY OF CHAR);
(*
* Read an INTEGER or a cardinal
* Skip blanks or control chars.
* Then read a sequence if digits as an INT or CARD
* (INTS can have a leading plus or minus sign)
* termch is the CHAR following the digits.
* Done = TRUE means a number was read.
*)
PROCEDURE ReadInt (VAR x:INTEGER);
PROCEDURE ReadCard (VAR x:CARDINAL);
(* Write a CHAR to the output connection *)
PROCEDURE Write; (ch:CHAR);
(* Start a new line *)
PROCEDURE WriteLn;
(* Write a string to the output connection *)
PROCEDURE WriteString (s: ARRAY OF CHAR);
END InOut.
DEFINITION MODULE Streams;
TYPE
     STREAM; ( (*Opaque Stream TYPE *)
VAR
     eolc: CHAR; (*Ths end of line marker *)
(*
* Connect a stream to an already open file
* identified by filenum.
* Set ws parameter TRUE to connect a WORD STREAM
*)
PROCEDURE Connect
(VAR s:STREAM; filenum :CARDINAL; ws:BOOLEAN);
(*
* Disconnect a stream from a file.
* Set closefile parameter to true to automaticaly
* close the file
*)
PROCEDURE Disconnect
          (VAR s: STREAM; closefile:BOOLEAN);
(*
* Are we at the end of a STREAM?
* TRUE --→ End Of Stream
```


Scanned with OKEN Scanner

```
*)
PROCEDURE EOS (s:STREAM): Boolean;
(*Note ReadChar supplies the value OC to ch
at End Of Stream *)
(* Retreat to the begining of the STREAM *)
PROCEDURE Reset (s:STREAM);
(*
* Call EndWrite afte the last write to a STREAM
* to write the end marker
*)
PROCEDURE EndWrite(s:STREAM);
(* The read and write PROCEDUREs *)
PROCEDURE ReadChar(s:STREAM; VAR ch:CHAR);
PROCEDURE ReadWord(s:STREAM; VAR w:WORD);
PROCEDURE WriteChar(s:STREAM; ch:CHAR);
PROCEDURE WriteWORD(s:STREAM; w:WORD);
(* Find out where a STREAM is positioned *)
PROCEDURE GetPos
     (s:STREAM; VAR high, low: CARDINAL);

(*Move to a given point in a STREAM *)
PROCEDURE SetPos(s:STREAM; high, low: CARDINAL);
(* Note high and low together from a long card *)
END Streams.
DEFINITION MODULES Files;
(* create a file with the given name *)
PROCEDURE Create (VAR descriptor: CARDINAL;
     name: ARRAY OF CHAR; VAR reply: INTEGER);
(* open a file with the given name *)
PROCEDURE Lookup(VAR descriptor: CARDINAL;
     name: ARRAY OF CHAR; VAR reply:INTEGER);
(* change the name of a file *)
PROCEDURE Rename
     (new, old :ARRAY OF CHAR: VAR reply: INTEGER);
(* Close a file *)
PROCEDURE Close (descriptor:CARDINAL);
(*
* Read count elements from a file
* starting at block
*)
PROCEDURE ReadF
     (descriptor : CARDINAL; VAR buf :ARRAY OF WORD);
     block, count: CARDINAL; VAR reply :INTEGER);
(*
* Write count elements into a file
*       starting at block
*)
PROCEDURE WriteF
     (descriptor :CARDINAL ;VAR buf :ARRAY OF WORD;
          block, count: CARDINAL ; VAR reply: INTEGER);
END Files.
```


Scanned with OKEN Scanner

```
Definition Module Mathlib;
PROCEDURE sqrt      (x:REAL):REAL;
PROCEDURE exp       (x:REAL):REAL;
PROCEDURE ln        (x:REAL):REAL;
PROCEDURE sin       (x:REAL):REAL;
PROCEDURE cos       (x:REAL):REAL;
PROCEDURE arctan    (x:REAL):REAL;
PROCEDURE real      (x:INTEGER):REAL;
PROCEDURE entier    (x:REAL):INTEGER;
End Mathlib.
Definition Module Processes;
PROCEDURE StartProcesses (P:PROC;n:CARDINAL);
(* Start a sequential process with program P and
workspace of size n*)
PROCEDURE SEND (VAR s:SIGNAL);
(* One process waiting for s is resumed *)
PROCEDURE Awaited (s:SIGNAL):BOOLEAN;
(* Awaited(s) = 'at least one process waiting for s'*)
PROCEDURE Init (VAR s:SIGNAL);
(* Compulsory initialization *)
Definition Module Program;
TYPE CallResult =    (NormalReturn, ProgramHalt,
                      RangeError, SystemError,
                      FuntionError, StackOverflow,
                      IntegerError, DivideByZero,
                      AddressError, UserHalt,
                      CodeIOError, UserIOError,
                      InstructionError,
                      FloatingError, StringError,
                      StorageError, VersionError,
                      MissingProgram, MissingModule,
                      LibraryError,
                      NotMainProcess, DuplicateName);
TYPE CallMode = (Shared,Unshared);
TYPE ErrorMode = (SystemTrap,CallerTrap);

PROCEDURE Terminate (exception:CallResult);
PROCEDURE Call  (programNAME:ARRAY OF CHAR;
                 calltype          :CallMode;
                 errors            :ErrorMode):CallResult;
TYPE EnvMode = (AllCals,UnsharedCalls,FirstCall);
PROCEDURE SetEnvelope (init,term:PROC;mode:EnvMode);
End Program.
Definition MODULE RealInOut;
VAR Done:BOOLEAN;
PROCEDURE ReadReal      (VAR x:REAL);
PROCEDURE WriteReal     (x:REAL;n:CARDINAL);
PROCEDURE WriteRealOct  (x:REAL);
End RealInOut.
Definition MODULE Reals;
```


Scanned with OKEN Scanner

```
PROCEDURE ReadReal (t:TEXT;VAR s:REAL);
PROCEDURE WriteReal (t: TEXT;r:REAL);
                    n:CARDINAL;digits:INTEGER);
PROCEDURE Real ToSTR  (r:REAL;digits:INTEGER;
                       VAR r:ARRAY OF CHAR):BOOLEAN;
PROCEDURE StrToReal   (s:ARRAY OF CHAR;
                       VAR r:REAL):BOOLEAN;

End Reals.
Definition MODULE Screen;
PROCEDURE HomeCursor; (* Move the cursor to upper left *)
PROCEDURE ClearScreen; (* Erase from cursor to end of screen *)
PROCEDURE EraseLine; (* Erase from cursor to end of line *)
PROCEDURE GotoXY(x, y:CARDINAL);(* Move to column x, row y *)
End Screen.
Definition MODULE Strings;
TYPE STRING=ARRAYCO..80] OF CHAR;
PROCEDURE Assign (VAR source, dest:ARRAY OF CHAR);
PROCEDURE Insert (VAR  str:ARRAY OF CHAR;
                       inx:CARDINAL;
                       len:CARDINAL);
PROCEDURE Delete (VAR str:ARRAY OF CHAR;
                       inx:CARDINAL;
                       len:CARDINAL);
PROCEDURE Pos (substr, str: ARRAY OF CHAR):CARDINAL;
PROCEDURE Copy  (str:ARRAY OF CHAR;
                 inx:CARDINAL;
                 len:CARDINAL;
                 VAR result:ARRAY OF CHAR);
PROCEDURE Concat (s1, s2:ARRAY OF CHAR;
                  VAR result:ARRAY OF CHAR);
PROCEDURE Lenght (VAR str:ARRAY OF CHAR):CARDINAL;
PROCEDURE CompareStr (s1, e2:ARRAY OF CHAR):INTEGER;

End Strings.

Definition MODULE Terminal;

PROCEDURE Read       (VAR ch:CHAR);
PROCEDURE ReadLn     (VAR s:ARRAY OF CHAR);
PROCEDURE BusyRead   (VAR ch:CHAR);
PROCEDURE ReadAgain;

PROCEDURE Write        (ch:CHAR);
PROCEDURE WriteString (s:ARRAY OF CHAR);
PROCEDURE WriteLn;

End Terminal

Definition MODULE SystemTypes;

CONST MinInt     = -32768;
      MaxInt     = 32767;
      MaxCard    = 65535;
```


Scanned with OKEN Scanner

```
        AdrsPerWord  = 2;
        CharsPerWord = 2;
End SystemTypes.
Definition MODULE Texts;
TYPE TEXT; (* Opaque type *)
VAR input,output,console:TEXT; (* Predeclared text files *)
PROCEDURE EOT (t:TEXT):BOOLEAN;(* End of text read. *)
PROCEDURE EOL (t:TEXT):BOOLEAN;(* End of line read. *)
TYPE TextState = (TextOk, FormatError,FileError,ConnectError);
PROCEDURE TextStatus (t:TEXT):TextState;
TYPE TextHandler = PROCEDURE (TextState);
PROCEDURE SetTextHandler  (t:TEXT;handler:TextHandler);
PROCEDURE Connect         (VAR t:TEXT;f:FILE):TextState;
PROCEDURE Disconnect      (VAR t:TEXT;:TextState;
PROCEDURE Read            (t:TEXT;VAR ch:CHAR);
PROCEDURE ReadInt         (t:TEXT;VAR i:INTEGER);
PROCEDURE ReadCard        (t:TEXT;VAR c:CARDINAL);
PROCEDURE ReadLn          (t:TEXT;VAR s:ARRAY OF CHAR);
PROCEDURE ReadAgain       (t:TEXT);
PROCEDURE Write           (t:TEXT; ch:Char);
PROCEDURE WriteString     (t:TEXT; s:ARRAY OF CHAR);
PROCEDURE WriteLn         (t:TEXT; i:INTEGER;n:CARDINAL);
PROCEDURE WriteInt        (t:TEXT;c,n:CARDINAL);
PROCEDURE WriteLn         (t:TEXT);
Texts.

Definition MODULE UnitIO;
TYPE IOResultType = (INoError,      (* 0 *)
                     IHardErr,
                     IBaudUnit,
                     IBadMode,
                     ITimeOut,
                     ILostUnit,     (* 5 *)
                     ILostFile,
                     IBadTitle,
                     INoSpace,
                     INoFile,       (* 10 *)
                     IDupFile,
                     IFileOpen,
                     INotOpen,
                     IBadFormat,
                     IBufOflow);    (* 15 *)
PROCEDURE IOResult():IOResultType;
(* Return value indicating the result of the previous I/O operation *)
PROCEDURE UnitStatus (UnitNo:CARDINAL;
                      Results:ADDRESS;
                      Option:CARDINAL);
(* Return status of the specified unit
   --see UCSD Pascal manual for details *)
```


Scanned with OKEN Scanner

```
PROCEDURE UnitBusy (UnitNo:CARDINAL):BOOLEAN;
(* Return TRUE if the specified unit is waiting
   for an I/O operation to complete *)

PROCEDURE UnitRead (UnitNo   CARDINAL;
                    Buffer   :ADDRESS;
                    Index    :CARDINAL;
                    NBytes   :CARDINAL;
                    BlkNum   :CARDINAL;
                    FlagWd   :BITSET);

(* Read Bytes from I/O unit into buffer *)

PROCEDURE UnitWrite (UnitNo  :CARDINAL;
                     Buffer  :ADDRESS;
                     Index   :CARDINAL;
                     NBytes  :CARDINAL;
                     BlkNum  :CARDINAL;
                     FlagWd  :BITSET);
(* Write Bytes in Buffer out to I/O unit *)

End UnitIO.

Definition MODULE Standards;

TYPE ScanType = (ScanUntil,ScanWhile);

PROCEDURE MoveLeft (SrcAddr :ADDRESS;
                    SrcInx   CARDINAL;
                    DestAddr :ADDRESS;
                    DestInx  :CARDINAL;
                    NBytes   :CARDINAL);

(* Move bytes from Source to Destination starting with
   the first byte in Source *)
PROCEDURE MoveRight (SrcAddr  :ADDRESS;
                     SrcInx   :CARDINAL;
                     DestAddr :ADDRESS;
                     DestInx  :CARDINAL;
                     NBytes   :CARDINAL);

(* Move bytes from Source to Destination starting with
   the last byte in Source *)

PROCEDURE FillChar (DestAddr:ADDRESS;
                    DestInx:  CARDINAL;
                    NBytes   :CARDINAL;
                    FillVal   :CHAR);

(* Initialize bytes in DEST with the byte value FillVal *)

PROCEDURE Scan (NumChars :INTEGER;
                ForPast  :ScanType;
```


Scanned with OKEN Scanner

```
                    Target      :CHAR;
                    Source      :ADDRESS;
                    ScrInx      :CARDINAL):INTEGER;
(* Starting at Source, scan for Numchars characters until
   Target character is found.Return offset from Source*)

PROCEDURE Time (VAR Hi,Lo:CARDINAL;
(* Return 32-bit system clock value in Hi and Lo*)

PROCEDURE Alloc (VAR p:ADDRESS;words:CARDINAL);
(* Allocate space on top of heap*)

PROCEDURE Mark (VAR p:ADDRESS);
(* Save current heap position in p*)

PROCEDURE Release (VAR p:ADDRESS);
(* Cut heap back to position specified by p*)

PROCEDURE MemAvail():CARDINAL;
(* Return  words between stack and heap top*)

PROCEDURE PowerOfTen (e:CARDINAL):REAL;
(* Return 10 raised to the eth power*)

End Standards.

Definition MODULE Conversions;

PROCEDURE IntToStr     (i:INTEGER;
                        VAR s:ARRAY OF CHAR):BOOLEAN;

PROCEDURE StrToInt   (s:ARRAY OF CHAR;
                      VAR i: INTEGER):BOOLEAN;
PROCEDURE CardToStr (c:CARDINAL;
                      VAR s:ARRAY OF CHAR):BOOLEAN;

PROCEDURE StrToCard    (s:ARRAY OF CHAR;
                        VAR c:CARDINAL):BOOLEAN;

PROCEDURE HexToStr     (w:WORD;
                        VAR s:ARRAY OF CHAR):BOOLEAN;

PROCEDURE StrToHex     (s:ARRAY OF CHAR;
                        VAR w:WORD):BOOLEAN;

End Conversions.

Definition MODULE BlockIO;

TYPE
      FileName  = ARRAY [0..39] OF CHAR;(*UCSD format string*)
      CloseType = (Normal,Lock,Purge,Crunch);
      FILE      = ARRAY [0..30] OF WORD;
```


Scanned with OKEN Scanner

```
(* Note that INTEGER parameters are to be
   used as CARDINAL. They are declared as
   INTEGER to match UCSD operating systems declarations exactly *)

PROCEDURE InitFile(VAR f:FILE);
(* Initialize FILE variabile...must be done before any other routines
   can be called.*)

PROCEDURE Reset(VAR f:FILE;VAR fn:FileName);
(* Open existing file.*)
PROCEDURE Rewrite(VAR f:FILE;VAR fn:FileName);
(* Open new file.*)

PROCEDURE Close(VAR f:FILE;ftype:CloseType);
(* Close file and update directory...*)

      Normal        Leave if opened with Reset, remove if
                    opened with Rewrite.
      Lock          Save permanent entry in directory.
      Purge         Remove entry from directory.
      Crunch        Save permanent entry, truncate
                    file at current file position.*)

PROCEDURE BlockRead(VAR f:FILE;buf:ADDRESS;byte index:INTEGER;
                          nblocks, startblock:INTEGER):INTEGER;
(* Read nblocks of the file into memory*)

PROCEDURE BlockWrite(VAR f:FILE;buf:ADDRESS;byte index:INTEGER;
                          nblocks,startblock:INTEGER):INTEGER;
(* Write nblocks of the file from memory*)

End BlockIO.

Definition MODULE Bits;

PROCEDURE LoadByte(base:ADDRESS;offset:CARDINAL):CARDINAL;
(* Load byte from byte address base[offset]*)

PROCEDURE StoreByte(base:ADDRESS:offset,ValueStore:CARDINAL);
(* Store byte at byte address base[offset]*)

PROCEDURE LoadField (VAR        w:WORD;
                     NumberOfBits:CARDINAL;
                     RightMostBit:CARDINAL):CARDINAL;
(* Load specified bit field from word w.*)

PROCEDURE StoreField (VAR       w:WORD;
                      NumberOfBits:CARDINAL;
                      RightMostBit:CARDINAL;
                      ValueToStore:CARDINAL);
(* Store specified bit field into word w.*)
```


Scanned with OKEN Scanner

End Bits.

```
Definition MODULE ASCII;

CONST
    nul = 00C; soh = 01C; stx = 02C; etx = 03C;
    eot = 04C; enq = 05C; ack= 06C; bel = 07C;
    bs  = 10C; ht  = 11C; If  = 12C; vt  = 13C;
    ff  = 14C; cr  =15C;  so  = 16C; si  = 17C;
    dle = 20C; dc1 = 21C; dc2 = 22C; dc3 = 23C;
    dc4 = 24C; nak = 25C; syn = 26C; etb = 27C;
    can = 30C; em = 31C;  sub = 32C; esc = 33C;
    fs=34C; gs=35C; rs=36C; us=37C;
    del = 177C;
```

Scanned with OKEN Scanner

BIBLIOGRAFIE

1. AHO, A.; HOPCROFT, J.; Ullman, J., *Data Structures and Algoritms*, Addison-Wesley Publishing Company, 1983.

2. DAVIDOVICI, A.; BĂRBAT, B., *Limbaje de programare în timp real*, Editura Tehnică, Bucureşti, 1986.

3. CHRISTIAN, K., *A Guide to MODULA-2*, Springer-Verlag, New York, Inc., 1986.

4. DIJKSTRA, E., *Solution of a problem in concurent programming control*, C.A.C.M, 8, 8 (1965).

5. FLOYD, R., *Assigning meaning to programs*, Mathematical Aspects of Computer Science, *19*(1967).

6. GRIES, D., *The Science of Programming*, Springer-Verlag, New York, Inc., 1986.

7. GLEAVES, R., *MODULA-2 for Pascal Programmers*, Springer-Verlag, New York, Inc., 1984.

8. GHEZZI, C; JAZAJERI, M., *Programming Language Concept*, John Wiley& Sons, New York, 1982.

9. HOROWITS, E., *Fundamentals of Programming Languages*, Computer Science Press, 1982.

10. HOWARD, J., *Proving monitors*, CACM, *19*, *5*, 1976.

11. PARNAS, D., *A tehnique for software module specification*, CACM, *15*, *5*, (1976).

12. THALMAN, D, *MODULA-2 An Introduction*, Springer-Verlag, New York, Inc., 1982.

13. WIENER, R.; SINCOVEC, R., *Software Engineering With ADA and Modula-2* John Wiley & Sons, New York, 1984.

14. WIRTH, N., *MODULA: A language for modular programming*, Software Practice and Experience, *7*, *1*, jan-feb. (1977).

15. WIRTH, N., *Programming in MODULA-2*, Springer-Verlag, New York, Inc., 1982.

Scanned with OKEN Scanner

PARTEA a III-a

PROGRAMARE ÎN PROLOG

CAPITOLUL 1.

INTRODUCERE ÎN PROLOG

1.1. Introducere

Limbajul PROLOG (*PRO*grammation en *LOG*ique) a fost elaborat la Universitatea din Marsilia, în jurul anului 1970, ca un instrument pentru programarea și rezolvarea de probleme ce implicau reprezentări simbolice de obiecte și relații dintre obiecte. Ca și LISP, PROLOG este un limbaj cu fundament matematic.

Limbajul LISP se bazează pe calculul lambda și este specializat în definirea de funcții. PROLOG — constituit pe un subset al logicii clasice — permite definirea și prelucrarea relațiilor, din care motiv deține un loc aparte în categoria limbajelor de programare.

Primul interpretor PROLOG a fost scris în limbajul FORTRAN de către echipa lui A. Colmerauer, care a lansat ideea utilizării logicii de ordinul întîi și demonstrarea automată a teoremelor, în scopul elaborării unui limbaj de înțelegere și analiză a limbajului natural. În prezent există mai multe versiuni ale limbajului PROLOG pentru diverse tipuri de calculatoare. PROLOG a fost inclus de Japonia ca limbaj de bază, în planul de dezvoltare a sistemelor de generația a cincea.

Limbajele de programare clasice (FORTRAN, BASIC, COBOL, PL/1, PASCAL etc.) sînt în esență limbaje algoritmice (sau imperative), în sensul că programele scrise în aceste limbaje cuprind comenzi ce descriu pas cu pas cum se obține rezultatul final. Ele cuprind și părți descriptive, dar într-o măsură foarte redusă. De exemplu, expresiile aritmetice pot fi considerate ca mici programe descriptive ce descriu modul în care se obține valoarea de calculat.

Scanned with OKEN Scanner

Limbajele de nivel înalt sînt mai ușor de învățat și de utilizat decît limbajele de asamblare pentru că sînt mai descriptive. Cu cît limbajul este mai descriptiv, cu atît este mai eficient în elaborarea de programe corecte, apropiate de specificațiile de definiție ale programului, care indica ce trebuie făcut și nu cum trebuie făcut.

Limbajul PROLOG are un pronunțat caracter descriptiv: un program PROLOG este în primul rînd o colecție de definiții ce descriu relații sau funcții de calculat. Execuția programului descriptiv constă în utilizarea definițiilor pentru a găsi o relație sau un obiect corespunzător unor specificații date. Soluția problemei nu se mai vede ca o execuție pas cu pas a unei secvențe de instrucțiuni. Modul de execuție al programului depinde în primul rînd de definițiile relațiilor sau funcțiilor, de inferența relațiilor și, de asemenea, de informația de control explicită furnizată de programator.

Limbajul PROLOG poate fi considerat ca un prim pas în abordarea noului stil de programare — programarea logică [5].

În prezent el are un cîmp foarte larg de aplicare, și anume: baze de date relaționale; inteligență artificială; logică matematică, demonstrare de teoreme; sisteme expert; inginerie arhitecturală; analize biochimice; rezolvare de probleme abstracte sau ecuații simbolice etc.

PROLOG constituie un limbaj conversațional, în sensul că utilizatorul se află la terminal într-un permanent dialog cu calculatorul pentru construirea programului sau obținerea rezultatelor. Programatorul specifică fapte și relații despre obiecte, dar sistemul PROLOG găsește, prin inferența relațiilor, noi relații dintre obiecte, uneori nebănuite chiar de programator.

În acest capitol se va descrie mecanismul de bază al limbajului PROLOG [1], ce a fost implementat sub sistemele de operare DECSYSTEM-10, UNIX și RT-11. Alte implementări prezintă unele diferențe relative la formatul relațiilor, interogarea bazei de date sau chiar la definirea variabilelor. Pentru detalii trebuie consultate manualele respective de implementare. Noțiunile de bază și modul de lucru al sistemului PROLOG este însă identic pentru toate implementările. Ca o exemplificare se descrie o variantă de micro-PROLOG implementată pe microcalculatorul ZX-SPECTRUM.

Există de asemenea o implementare a limbajului PROLOG sub sistemul de operare MIX, pentru minicalculatoarele românești de tip INDEPENDENT și CORAL, și anume interpretorul MIX-PROLOG elaborat de un colectiv de cercetători din ICSIT-TCI [12].

Menționăm că materialul prezentat nu-și propune o descriere riguroasă a limbajului PROLOG, ci caută să scoată în evidență modul de lucru și principiile programării în acest limbaj, ce sînt comune tuturor implementărilor existente.

Scanned with OKEN Scanner

1.2. Noțiunile de bază ale limbajului PROLOG

În principiu, programarea în PROLOG se reduce la următoarele: se specifică însușiri de obiecte și relații dintre obiecte (fapte); se specifică reguli (sau proceduri) ce descriu în mod generic relații dintre obiecte; se pun întrebări relativ la obiecte sau relații.

Mulțimea de fapte și reguli formează baza de date a programului PROLOG.

Sistemul PROLOG distinge două moduri de lucru: modul de intrare, în care se introduc fapte și reguli (se crează baza de date), și modul de interogare, în care se pun întrebări referitoare la obiecte sau relații (În această prezentare modul de interogare se indică prin șirul de caractere „? —".)

În continuare se vor detalia noțiunile de *fapt*, *întrebare* și *regulă* relative la limbajul PROLOG.

● *Fapte.* În limbajul PROLOG un fapt exprimă o însușire a unui obiect sau o relație între două sau mai multe obiecte. De exemplu, faptul că Ionescu este student se exprimă în PROLOG astfel:

```
student(ionescu).
```

Faptul că Dan este prietenul lui Radu se exprimă astfel:

```
prieten(dan, radu).
```

În PROLOG faptul se va numi și *predicat*, iar obiectele dintre paranteze — *argumente*. Se observă că elementul separator dintre argumente este virgula, iar spațiul după separator nu este semnificativ.

În descrierea relației a fost adoptat formatul prefix, adică se scrie mai întîi numele relației, urmat de lista de argumente între paranteze. Alte implementări ale limbajului PROLOG utilizează și alte forme. De exemplu, pentru relațiile binare se folosește formatul infix,

```
obiect-1   nume-relație   obiect-2,
```

iar în ceea ce privește relațiile unare se utilizează uneori formatul postfix:

```
obiect nume-relație.
```

În continuare se au în atenție următoarele convenții pentru scrierea relațiilor:

— se aplică formatul prefix;
— argumentele se separă prin virgulă;
— numele argumentelor și al relațiilor încep cu literă mică;
— relația se termină cu caracter final '.'.

Scanned with OKEN Scanner

Aceste convenții se întîlnesc în majoritatea implementărilor limbajului PROLOG.

Ordinea în care se scriu argumentele depinde de sensul relației. De exemplu, relația

șef(dan, radu)

exprimă faptul că Dan este șeful lui Radu și nu invers.

Sensul unei relații este stabilit de către programator și el trebuie să fie același pe tot parcursul programului. Pentru o bună înțelegere a programului se recomandă ca numele de relații și obiecte să corespundă cu interpretarea dorită de programator.

• *Întrebări.* În PROLOG formularea unei întrebări se exprimă ca o relație, cu deosebire că relația este precedată de șirul de caractere „? —".

Fie de exemplu întrebarea:

? — prieten(dan, radu).

Ea se referă la faptul dacă Dan este prieten cu Radu. În acest caz se consultă baza de date a programului PROLOG, iar dacă se identifică o relație care are același nume cu întrebarea și aceleași argumente se va da un răspuns afirmativ. Altfel se va răspunde „nu".

Fie de exemplu următoarea bază de date a programului PROLOG:

```
prieten(dan, radu).
prieten(dan, ion).
frate(ion, maria).
student(ion).
```

Următoarele întrebări puse sistemului PROLOG vor primi răspunsul ce este specificat după fiecare:

```
? — prieten(radu, ion).
nu
? — prieten(dan, radu).
da
? — frate(ion, radu).
nu
? — student(ion).
da
```

Evident că răspunsurile la întrebări depind exclusiv de conținutul și consistența bazei de date a programului PROLOG.

Întrebările precedente au caracter de confirmare, adică la întrebarea pusă se întoarce răspunsul „da" sau „nu". Pentru a obține informații din baza de date se pot folosi, în cadrul întrebărilor, obiecte generice numite *variabile*. O variabilă poate fi inițializată, caz în care desemnează un obiect sau o proprietate, sau neinițializată.

Scanned with OKEN Scanner

În majoritatea implementărilor limbajului PROLOG variabilele încep cu literă mare. De exemplu, dacă se dorește să se afle prietenul lui Dan se pune următoarea întrebare:

? — prieten(dan, X).,

unde *X* este o variabilă. De asemenea, orice nume de argument care începe cu literă mare este o variabilă. De exemplu:

? — prieten(dan, Prieten-dan).

Dacă se consideră baza de date de mai sus, în acest caz sistemul PROLOG va răspunde:

X = radu.

Căutarea în baza de date se oprește la prima concordanță dintre întrebare cu primul argument și relație cu primul argument.

Utilizatorul are două posibilități:

— să fie satisfăcut cu primul răspuns și să oprească procesul de căutare introducînd CR (carriage return);

— să continue căutarea, introducînd caracterul „;" urmat de CR. De pildă, la întrebarea de mai sus programul PROLOG va răspunde cu:

X = ion.

Dacă utilizatorul introduce în continuare „;" și CR, atunci răspunsul la întrebarea pusă mai sus va fi „nu", ceea ce înseamnă că s-a ajuns la sfîrșitul bazei de date, fără să se satisfacă întrebarea.

Evident că variabila poate fi în orice poziție a listei de argumente. De exemplu, se poate întreba:

? — prieten(X, radu).

În alte implementări ale limbajului PROLOG, la o întrebare generică (cu variabile) se furnizează automat toate răspunsurile posibile, iar dacă se dorește un singur răspuns se folosește o întrebare specială.

Se pot pune întrebări complexe prin utilizarea operatorului „și" (conjuncția). În limbajul PROLOG acest operator se reprezintă în general prin virgulă (în alte implementări se scrie „and" sau „&").

Fie minibaza de date următoare:

```
prieten(dan, vlad).
prieten(dan, ion).
prieten(dan, mihai).
prieten(radu, george).
prieten(radu, ion).
```

Pentru a afla prietenul comun al lui Dan și Radu se pune următoarea întrebare:

? — prieten(dan, X), prieten(radu, X).

Virgula are rolul conjuncției „și"; ea desparte mai multe întrebări din cadrul unei secvențe de întrebări de la care se așteaptă un răspuns care să le satisfacă pe toate.

Scanned with OKEN Scanner

Cum lucrează sistemul PROLOG? În cazul exemplului de mai sus se caută mai întîi în baza de date un prieten al lui Dan, iar dacă se găsește (fie Vlad) se atribuie acest nume variabilei X. Cu variabila X astfel inițializată se parcurge baza de date, de la începutul ei pentru a găsi o relație „prieten (radu, vlad)".

Dacă nu se găsește (ceea ce este cazul) se continuă parcurgerea bazei de date de unde s-a rămas în cazul primei relații și se atribuie variabilei X valoarea „ion". Se parcurge din nou baza de date de la începutul ei, cu noua valoare a lui X pentru a găsi o relație „prieten (radu, ion)".

În acest caz această relație există, iar răspunsul final la întrebare va fi:

X = ion.

Sistemul PROLOG așteaptă pentru o nouă intervenție.

Dacă se răspunde cu „;" și CR, atunci se va continua cu prima relație pentru a găsi un nou prieten al lui Dan. În acest caz X va lua valoarea „mihai" și se va continua parcurgerea bazei de date de la începutul ei pentru a găsi o relație „prieten (radu, mihai)" ș.a.m.d.

În situația că se răspunde cu CR sistemul PROLOG așteaptă o nouă comandă.

Se observă că satisfacerea variabilelor se face de la stînga la dreapta. Locul în baza de date în care o variabilă a primit o valoare se marchează și se încearcă satisfacerea următoarei variabile din dreapta prin parcurgerca bazei de date de la începutul ei. În caz de insucces se revine la variabila imediată din stînga și se încearcă atribuirea unei noi valori prin parcurgerea bazei de date din poziția marcată. Acest procedeu de parcurgere a bazei de date în scopul evaluării unei întrebări cu mai multe variabile este cunoscut în PROLOG sub numele de *backtracking*, adică revenire înapoi. De aici rezultă că în cazul întrebărilor complexe, pentru o parcurgere eficientă a bazei de date, este necesar ca spre începutul întrebării să apară condițiile cu mai puține soluții de rezolvare.

• *Reguli.* În PROLOG regula este o descriere generală (fără date concrete) a obiectelor și relațiilor dintre obiecte. Din punct de vedere sintactic ea conține un antet și un corp, despărțite prin șirul de caractere „:—". Șirul de caractere „:—" are semnificația „dacă" (în alte implementări în loc de „:—" se folosește cuvîntul cheie 'if').

Regula este o definiție implicită a unei relații. Formatul ei este următorul:

relație :— relație-1, relație-2, relație-3....

Ea are semnificația următoare: „relația" specificată este adevărată dacă „relație-1", „relație-2", „relație-3"... sînt adevărate.

Scanned with OKEN Scanner

Baza de date a unui program PROLOG este formată din fapte și reguli. O intrare din baza de date, fapt sau regulă, se mai numește și *clauză*. Fie regula următoare:

frate (X, Y) : — bărbat(X), părinți (X, T, M), părinți (Y, T, M).

Ea exprimă faptul că X este fratele lui Y, dacă X este bărbat și dacă X, Y au aceiași părinți (T — tată, M-mamă), unde relația:

părinți(X, T, M)

arată că X are tata T și mama M.

Fie de exemplu baza de date următoare:

părinți(ion, luca, maria).
părinți(dan, luca, maria).
părinți(vlad, george, ioana).
bărbat(dan).
bărbat(ion).
bărbat(vlad).
frate(X, Y) : — bărbat(X),
părinți(X, T, M), părinți(Y, T, M).

Să punem sistemului PROLOG următoarea întrebare:

? — frate(dan, X).

Sistemul PROLOG procedează în felul următor:

— întrebarea corespunde antetului regulii „frate(X, Y)", în care X se inițializează cu valoarea „dan". Variabila Y din regulă corespunzătoare lui *X* din întrebare rămîne neinițializată;

— prima relație din corpul regulii: „bărbat(dan)" este găsită în baza de date și marcată ca adevărată;

— a doua relație din corpul regulii devine:

părinți(dan, T, M),

unde T și M sînt variabile neinițializate. Sistemul PROLOG caută în baza de date, găsește relația:

părinți(dan, luca, maria)

și inițializează variabilele T și M cu respectiv „luca", și „maria".

— a treia relație din corpul regulii devine:

părinți(Y, luca, maria)

și se caută în baza de date o corespondență în acest sens. Se găsește:

părinți(ion, luca, maria);

deci, sistemul va răspunde la întrebarea precedentă cu:

„X = ion".

Regulile sînt necesare în definirea relațiilor recursive, așa cum se va vedea în paragrafele următoare.

Se remarcă faptul că notația PROLOG : — este inversă notației relației „implică" din logică. Astfel $a \rightarrow b$ (în logică) se scrie în PROLOG b: —a, ceea ce se citește: pentru a demonstra *b* trebuie demonstrat *a*.

Scanned with OKEN Scanner

Aceasta se explică prin faptul că în parcurgerea și căutarea în baza de date se aplică principiul revenirii înapoi *(backtracking)*. Capul regulii nu comportă decît un singur literal (predicat), pe cînd corpul poate fi o conjuncție de predicate.

Căutarea unei soluții pentru o întrebare poate fi considerată drept demonstrarea unei teoreme. În termeni de demonstrare automată se utilizează o procedură de unificare, ce constă în găsirea valorilor pe care trebuie să le ia variabilele conținute în două relații pentru a le face să coincidă, iar cuplurile (variabilă, valoare) găsite formează mulțimea de substituții. De exemplu, termenii frate(dan, X) și frate (dan ion) se unifică prin substituția X = ion.

1.3. Elementele limbajului PROLOG

Programele PROLOG sînt formate din *termeni*. Un *termen* poate fi o *constantă*, o *variabilă* sau o *structură*. Termenul se scrie ca o secvență de caractere.

Caracterele admise în formarea unui termen sînt următoarele*:

— litere mari: *A B C D ... X Y Z*;

— litere mici: *a b c d ... x y z*;

— cifre 0 1 2 3 4 5 6 7 8 9;

— caractere speciale: + — * / \ < > = ' " : . ? ^ ω # $ & —.

Se va specifica pentru fiecare tip de termen caracterele admise pentru formarea numelui.

• *Constanta* desemnează un obiect sau o relație.

Există două categorii de constante: *atomi* și *întregi*. Numele folosite în paragraful anterior, adică prieten, dan, radu etc., precum și secvențele de caractere ?— și :— sînt atomi. Se deosebesc de asemenea două tipuri de atomi: formați numai din litere și cifre (alfanumerici) și, respectiv, numai din caractere speciale.

Din punct de vedere logic atomul este o entitate indivizibilă.

Dacă un atom este închis între apostrofuri('), atunci poate conține orice caracter special. Face excepție de la această regulă caracterul '—' care, din motive de înțelegere mai ușoară a numerelor, poate fi inclus în interiorul unui atom fără a închide atomul între apostrofuri. Exemple de atomi: ab; =+; nume-1; popescu__ion; 'popescu _ion'.

---

* Sistemul PROLOG recunoaște tot setul de caractere ASCII tipăribile și netipăribile. Unele caractere netipăribile au ca efect diverse acțiuni — saltul la o nouă linie, emiterea unui sunet la terminal (bell) etc. — , iar alte caractere tipăribile au semnificații speciale pentru sistemul PROLOG — de exemplu, parantezele rotunde ( ) pentru închiderea argumentelor unei relații sau structuri, virgula ca separator etc.

Scanned with OKEN Scanner

În ultimul exemplu s-a folosit caracterul '__' și nu '—' pentru a forma un atom și în acest caz a trebuit inclus numele atomului între apostrofuri. Lungimea șirului de caractere ce specifica un atom este limitată în general la 255 de caractere.

Întregii (integers) se folosesc pentru a reprezenta numere utilizate în expresii aritmetice. Constanta întreg este formată numai din cifre; de exemplu:

0 1 435 726 1434 65535 etc.

Valoarea maximă a întregului depinde de capacitatea de calcul a mașinii pe care se află sistemul PROLOG.

Există versiuni ale limbajului PROLOG în care pot fi utilizate și numere negative sau numere raționale cu diverse precizii. În general, aplicațiile PROLOG nu necesită calcule cu numere foarte mari sau numere raționale.

• *Variabila* este asemănătoare unui atom alfanumeric, dar începe cu literă mare sau cu caracterul '__'. Ea este un nume generic pentru un obiect. Exemple de variabile:

X Y Nume__45.

Semnul '__' utilizat singur formează așa-numita variabilă anonimă. De pildă el se folosește într-o întrebare, atunci cînd nu ne interesează răspunsul explicit: ? —prieten(__, radu).

În întrebarea de mai sus interesează numai dacă Radu are vreun prieten, dar nu explicit numele acelui prieten.

Lungimea numelui unei variabile este limitată în general la 255 de caractere.

• *Structura* este o expresie de forma f(a1, a2, ..., an), unde *f* este numele *functorului* (structurii), iar *a1*, *a2*, ..., *an* sînt termeni. Termenii *a1*, *a2*, ..., *an* se mai numesc *argumentele* functorului *f* sau *componentele* structurii, iar *n* se numește *aritatea* functorului.

Predicatele utilizate pînă acum sînt functori de structură, iar argumentele unei relații sau reguli, componentele unei structuri. Exemplu: persoană(nume, prenume, data__nașterii).

De asemenea, *persoană (nume, prenume,data__nașterii(zi, lună, an))* este tot o structură în care componenta *data__nașterii* este un termen de tip structură.

Structurile permit mai buna organizare a datelor complexe, deoarece prin intermediul lor un grup de informații poate fi tratat cu un singur obiect.

1.4. Operatori

Deoarece programatorii sînt familiarizați cu expresiile aritmetice este convenabil să se scrie anumiți functori ca operatori.

Scanned with OKEN Scanner

De exemplu, structura

$$+(x, -(y, z))$$

se citește mai ușor sub forma:

$$x+y-z.$$

Se observă că modul de scriere al structurii în care s-au utilizat operatorii + și — este mai simplu de utilizat.

Fiecare operator se caracterizează prin poziție, precedență și asociativitate.

Din punctul de vedere al poziției, un operator poate fi între argumente (infix), înainte de argument (prefix) sau după argument (postfix).

Precedența indică ce operator trebuie executat mai întîi.

În PROLOG fiecare operator aparține unei clase de precedență, de la 1 la 255. Cu cît precedența este mai mare, cu atît ea este mai apropiată de 1. În PROLOG operatorii * și / au precedență mai mare decît + și —, astfel că expresia x + y * z este echivalentă cu x + (y*z). Precedența se utilizează pentru a elimina ambiguitatea expresiilor atunci cînd nu se folosesc paranteze.

În limbajul studiat asociativitatea operatorilor este cea normală, adică de la stînga la dreapta, în sensul că operatorii de aceeași precedență se execută în ordinea întîlnirii lor. Astfel, expresia x*y/z este echivalentă cu (x*y)/z.

Vom prezenta pe scurt operatorii limbajului PROLOG.

Operatorul „=" este infix. El are rol de comparare cu implicații de atribuire. Fie expresia de forma X=Y, unde X și Y sînt doi termeni. Pentru a evalua această expresie, sistemul PROLOG procedează după următoarele reguli:

— dacă X este o variabilă neinițializată, iar Y — orice obiect, atunci egalitatea este adevărată, iar X este inițializat cu Y;

— întregii și atomii sînt egali cu ei înșiși. De exemplu,

10=10 este adevărat
10=12 este fals
dan = dan este adevărat

— două structuri sînt egale dacă au același nume și aceleași componente. Variabilele neinițializate dintr-o structură se inițializează cu valoarea celor din structura din egalitate. De pildă, din egalitatea:

persoană(ionescu, ion) = persoană(X, ion),

X ia valoarea „ionescu";

— dacă ambele variabile sînt neinițializate, atunci egalitatea devine adevărată, iar variabilele vor partaja aceeași locație în memorie, astfel că în momentul în care o variabilă se inițializează, se va inițializa automat și cealaltă variabilă.

Pentru semnul de neegalitate se folosește operatorul „\=".

Scanned with OKEN Scanner

Expresia X\=Y este adevărată dacă X=Y este falsă.

În principiu operațiile aritmetice constau în comparări și efectuări de calcule.

Operatorii de comparare relaționali sînt următorii: = (egalitate); \=(inegalitate); <(mai mic); >(mai mare); =<(mai mic sau egal); >= (mai mare sau egal).

Pentru efectuarea de calcule, orice sistem PROLOG posedă următorii operatori: + (adunare); — (scădere); * (înmulțire); / (împărțire); mod (restul împărțirii).

Operatorul PROLOG ce indică evaluarea unei expresii aritmetice și atribuirea rezultatului este „is". De exemplu, X is Y*Z arată că se efectuează calculul Y*Z, iar rezultatul se atribuie variabilei Y.

1.5. Lista

O structură poate fi mai ușor înțeleasă dacă se reprezintă sub formă de arbore, în care numele structurii este rădăcina, iar componentele sînt ramurile. De exemplu, structura:

persoană(nume, prenume, data_nașterii (zi, lună, an))

se poate reprezenta ca în fig. III.1.

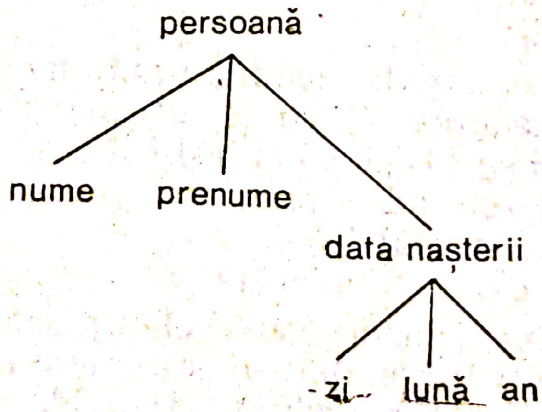


Fig. III.1.

O structură utilizată în aproape toate limbajele nenumerice este *lista* — secvență ordonată de elemente de lungimi diferite.

Elementele listei pot fi atomi, structuri sau alte liste.

În PROLOG lista este o structură particulară.

Orice listă are un „cap de listă" și o coadă. Face excepție lista vidă, care nu conține nici un element. Totodată, coada listei poate să fie lista vidă.

Scanned with OKEN Scanner

Lista vidă se reprezintă prin [ ]. Capul de listă și coada sînt componentele numelui de structură reprezentat prin caracterul „.“. Astfel, lista formată din atomii x, y, z se scrie după cum urmează: .(x,.(y,.(z,[]))). Arborele ei este reprezentat în fig. III.2.

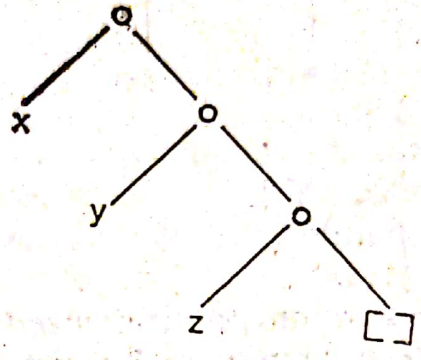


Fig. III.2.

Punctul (.) se poate defini ca operator, caz în care lista se scrie: x.(y.(z.[ ]))

Dacă se folosește și proprietatea de asociativitate la dreapta a operatorului „.“, atunci lista se scrie astfel: x.y.z.[ ]

Lista de mai sus se poate reprezenta ca în fig. III.3.

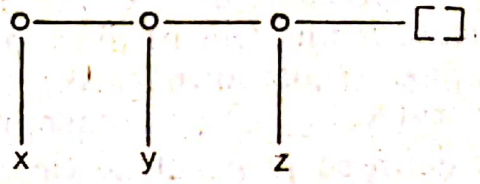


Fig. III.3.

În PROLOG reprezentarea listei este mult simplificată, în sensul că elementele listei se separă prin virgulă și sînt închise între paranteze drepte[,]. Astfel, în PROLOG lista de mai sus se scrie: [x, y, z].

Structura următoare este de asemenea o listă:

[x, y, z,[s, t, u]]

Ea se reprezintă în figura III.4.

În lista de mai sus capul de listă este x, iar coada listei, [y, z,[s,t,u]]. Lista vidă [ ] nu are nici cap nici coadă.

În PROLOG există un operator, reprezentat prin bara verticală (|), care separă o listă în „cap de listă“ și „coadă“.

Astfel, prin notația X|Y se inițializează X cu capul de listă, iar Y cu coada listei. Fie, de exemplu, predicatul:

p([x, y, z]).

Scanned with OKEN Scanner

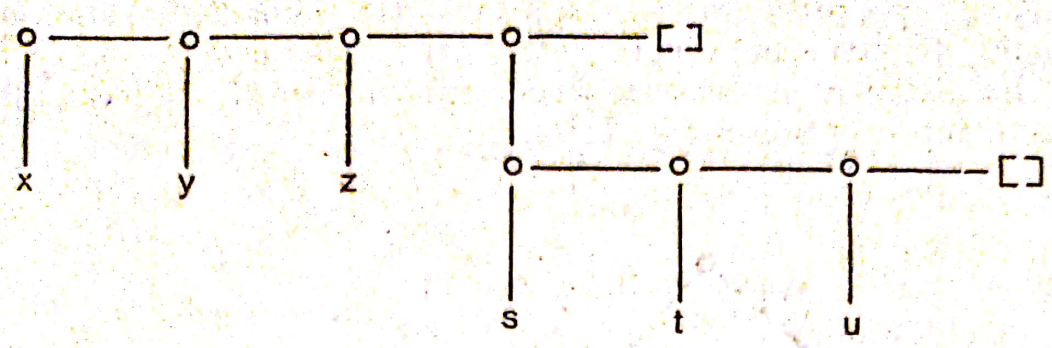


Fig. III. 4.

La întrebarea:

? — p([X | Y]).

sistemul PROLOG răspunde:

X = x Y = [y, z].

Exemplul 1. Vom scrie un predicat *membru*(*X*, *Y*) care ia valoarea „adevărat" dacă *X* este o componentă a listei *Y*. Acest exemplu pune în evidență procesul de recurență, foarte des întîlnit în problemele de parcurgere a listelor și în multe alte programe PROLOG.

X aparține listei *Y* dacă *X* este cap de listă sau un element din coada listei. Prima condiție se scrie în PROLOG prin regula următoare:

membru(*X*, [*Y* | _]) : _ X = Y.

sau mai simplu:

membru(X, [X | _]).

Variabila anonimă '_' reprezintă orice coadă de listă. Dacă *X* nu este cap de listă, atunci poate fi un element din coada listei, iar acest lucru se exprimă recursiv prin regula următoare:

membru(X, [_ | Y]) : — membru(X, Y).

Cele două reguli definesc procesul de căutare a elementului *X* în lista *Y*, adică *X* poate fi cap de listă, sau *X* poate fi cap de listă din coada listei *Y* ș.a.m.d. În momentul în care coada listei se reduce la lista vidă procesul de căutare se oprește, iar predicatul *membru* ia valoarea „fals". Mai jos se prezintă un exemplu de program PROLOG ce utilizează predicatul *membru*:

```
membru(X, [X | _]).
membru(X, [_ | Y]) : — membru(X, Y).
? — membru(5, [2, 3, 5, 7]).
da
? — membru(b, [a, e, f, g]).
nu
```

Exemplul 2. Se va scrie în PROLOG un program ce compară alfabetic cuvinte (sau atomi). Fie *comp*(*X*, *Y*) acest predicat, care ia valoarea „adevărat" dacă *X* este „mai mic" din punct de vedere alfabetic decît *Y*. De exemplu, comp(caiet, carte) furnizează răspunsul „da",

Scanned with OKEN Scanner

iar comp(carte, abecedar) răspunde cu „nu". Compararea cuvintelor se face literă cu literă pînă se ajunge la o condiție de neegalitate. Pot exista mai multe cazuri:

— se ajunge la sfîrșitul primului cuvînt, fără a termina parcurgerea celui de-al doilea; în acest caz *comp* răspunde cu „da";

— primul caracter din primul cuvînt este deja „mai mic" decît primul caracter din al doilea cuvînt — de exemplu, *comp(carte,pagină)*. În acest caz comp răspunde „da";

— primul caracter din primul cuvînt este „egal" cu primul caracter din al doilea cuvînt — de exemplu, *comp(casă,capă)*. În această situație se continuă compararea pe restul cuvintelor obținute înlăturîndu-se primul caracter ș.a.m.d.;

— dacă se ajunge simultan la sfîrșitul celor două cuvinte fără a întîlni condiția de neegalitate, înseamnă că cuvintele respective sînt identice și în acest caz *comp* furnizează răspunsul „nu";

— se ajunge la sfîrșitul celui de-al doilea cuvînt fără a termina parcurgerea primului — de exemplu, *comp(aparat,apa)*. În acest caz se furnizează răspunsul „nu".

Cuvintele vor fi reprezentate printr-o listă de caractere, iar caracterele prin codurile ASCII corespunzătoare. Fie predicatul *name(X, L)*, care transformă cuvîntul *X* în lista *L*. De exemplu,

```
? — name(abc, L).
L = [97, 98, 99].
```

Programul PROLOG pentru definirea predicatului *comp* este următorul:

```
comp(X, Y): — name(X, L), name(Y, M), compară(L, M).
compară([ ], [_ | _]).
compară([X | _], [Y | _]): — X<Y.
compară([A | X], [A | Y]) : — compară(X, Y).
```

Prima regulă transformă cuvintele X și *Y* respectiv în listele *L* și *M*. Apoi se compară listele *L* și *M* ale căror elemente sînt acum numere întregi (codurile ASCII corespunzătoare caracterelor). A doua regulă rezolvă, cu succes, cazul în care primul argument lipsește (rezultă lista vidă). Următoarele două reguli compară, recursiv, elementele (numere întregi) listelor *L* și *M*.

1.6. Căutarea în baza de date

Sistemul PROLOG examinează relațiile și regulile din baza de date în ordinea în care au fost furnizate de programator; ca urmare răspunsurile la întrebări sau alte evaluări, depind de această ordine.

Scanned with OKEN Scanner

Fie următorul program PROLOG:

```
p(a).
p(b).
p(c).
q(s).
q(t).
pereche(X, Y) : — p(X), q(Y).
```

Dacă se pune întrebarea:

```
? — pereche(X, Y).
```

iar după fiecare răspuns furnizat de sistemul PROLOG se răspunde cu ';' și CR, atunci se vor obține în ordinea de mai jos următoarele răspunsuri:

```
X = a   Y = s
X = a   Y = t
X = b   Y = s
X = b   Y = t
X = c   Y = s
X = c   Y = t
```

Se observă că pentru o valoare a lui X se epuizează mai întîi toate valorile ce le poate lua Y, după care se caută o nouă valoare pentru X în baza de date și toate valorile ce le poate lua Y ș.a.m.d.

Acest mod de parcurgere a bazei de date, denumit *backtracking* (revenire înapoi), a fost prezentat și în paragraful 2.

O regulă recursivă poate genera o infinitate de răspunsuri. De exemplu,

```
este_întreg(0).
este_întreg(N) : — este_întreg(M), N is M + 1.
```

La întrebarea:

```
? — este_întreg(N).
```

se va obține răspunsul:

```
N = 0,
```

iar dacă se răspunde de fiecare dată cu ';' și CR, atunci în secvență se vor obține răspunsurile:

```
N = 1
N = 2
N = 3
. . . .
```

Într-un program PROLOG ordinea definițiilor și a regulilor recursive nu este întîmplătoare.

Dacă în exemplul de mai sus se folosește definiția „este_întreg(0)" după regulă, atunci sistemul PROLOG lucrează fără a da nici un răspuns, pînă cînd își termină execuția din lipsă de spațiu în memorie.

Scanned with OKEN Scanner

Într-adevăr, la întrebarea este_întreg(N) se generează subproblema este_întreg(M). Pentru rezolvare se parcurge baza de date de la începutul ei, deci se ia în considerare tot regula recursivă și se generează aceeași subproblemă ș.a.m.d.

Din motivul menționat se recomandă ca definițiile să preceadă, în general regulile. Trebuie avut în vedere, așa cum s-a mai spus, că de ordinea definițiilor și regulilor depinde și ordinea răspunsurilor furnizate utilizatorului.

1.7. Tăietura

Mecanismul „tăieturii" se folosește în PROLOG din motive de eficiență și anume:

— programul se execută mai repede deoarece se exclud anumite parcurgeri ale bazei de date;

— programul ocupă mai puțin spațiu deoarece se memorează mai puține puncte de reluare *(backtracking)* pentru examinarea bazei de date.

Din punct de vedere sintactic, tăietura este un *predicat* fără argumente reprezentat prin semnul exclamării, '!'.

Fie exemplul următor:

```
p: — a, b, !, c, d,
a: — x, y.
b: — z, w.
c: — s, t.
d: — u.
```

Se presupune că pentru rezolvarea unei întrebări relative la *p* se parcurge baza de date astfel că *a* ia valoarea *x* iar *b* ia valoarea *z*. În momentul în care se depășește tăietura (semnul exclamării) se șterg toți pointerii de revenire înapoi pentru a atribui alte valori lui *a* și *b*. Să presupunem că se continuă parcurgerea bazei de date și *c* ia valoarea *s* iar *d* ia valoarea *u*.

Dacă valoarea *u* pentru *d* nu duce la rezultatul așteptat atunci se revine înapoi (backtracking) și se atribuie lui *c* valoarea *t*, după care se continuă cu *d*, atribuindu-se valoarea *u*.

Considerăm că și în acest caz se ajunge la un rezultat fals. În acest caz ar trebui revenit înapoi pentru a încerca o nouă valoare pentru *b* și anume *w*. Acest lucru nu mai este posibil, din cauza tăieturii. Deci variabilele care au primit valori înainte de tăietură rămîn cu aceste valori pentru toate încercările de după tăietură.

Scanned with OKEN Scanner

În general, programatorul folosește o tăietură în următoarele cazuri:
— să informeze sistemul PROLOG că a găsit regula corespunzătoare și nu trebuie să mai facă alte încercări;
— să informeze sistemul PROLOG că regula găsită duce la un rezultat fals și trebuie să abandoneze imediat orice căutare.

Al doilea caz se aplică împreună cu predicatul „fail". De exemplu, se presupune că se construiește un program PROLOG care să afișeze retribuția personalului muncitor dintr-o întreprindere. În acest caz dacă persoana specificată în întrebare este economist sau inginer, atunci sistemul PROLOG trebuie să abandoneze prelucrarea.

Acest lucru se poate exprima în PROLOG astfel:

```
retribuția(X): — economist(X), !, fail.
retribuția(X): — inginer(X), !, fail.
retribuția(X): — . . .(afișarea retribuției).
```

Tăietura este un mecanism de programare nu întotdeauna clar. De aceea, de mai multe ori se caută să se evite prin utilizarea predicatului „not". Sintaxa lui este următoarea:

```
not(x)
```

și este „adevărat" numai dacă *x* este „fals".

Fie de exemplu, predicatul *suma*(*N*,*X*), care inițializează pe *X* cu suma numerelor naturale de la 1 la *N*. Definiția lui în PROLOG este următoarea:

```
suma(1, 1): —!.
suma(N, R): — M is N —1, suma(M, T), R is T+N.
```

În acest caz întrebarea:

```
? — suma(5, X)
```

va determina răspunsul:

```
X = 15
```

deoarece $1 + 2 + 3 + 4 + 5 = 15$.

1.8. Intrări-ieșiri

După cum s-a arătat, la întrebările puse sistemului PROLOG, acesta răspunde astfel:

```
variabila = răspuns.
```


Scanned with OKEN Scanner

De multe ori răspunsul este un mesaj, adică o înșiruire de cuvinte. Un mod de reprezentare a mesajelor în PROLOG este lista.

Fie de exemplu o bază de date cu mai multe răspunsuri ce sînt numerotate:

```
răspuns(1, ['eroare', 'de', 'sintaxă']).
răspuns(2, ['fișier', 'inexistent']).
răspuns(3, ['comanda', 'este', 'incorectă']).
```

Prin întrebarea:

```
? — răspuns(3, X)
```

se dorește mesajul numărul 3. În acest caz sistemul PROLOG va răspunde:

```
X = [comanda, este, incorectă].
```

Se observă că forma răspunsului nu este corespunzătoare limbajului obișnuit.

Constanta „șir de caractere" se reprezintă în PROLOG prin codul ASCII al caracterelor. De exemplu, mesajul:

„eroare de sintaxă"

se reprezintă intern sub formă de listă astfel:

[101, 114, 111, 97, 114, 101, 32, 100, 101, 32, 115, 105, 110, 116, 97, 120, 97].

Se consideră că mesajele se scriu sub formă de constante șir:

```
răspuns(1, „eroare de sintaxă").
răspuns(2, „fișier inexistent").
răspuns(3, „comanda este eronată").
```

În această situație, dacă se pune întrebarea:

```
? — răspuns(1, X),
```

atunci sistemul PROLOG răspunde cu lista formată din codul șirurilor de caractere ale mesajului:

```
X = [101, 114, 111, ....].
```

Evident, acest răspuns nu este deloc satisfăcător.

Sistemul PROLOG pune la dispoziția utilizatorului mai multe predicate de intrare/ieșire în vederea schimbului de informații într-un format inteligibil.

Scanned with OKEN Scanner

Scrierea termenilor

Pentru scrierea unui termen la display sau terminal se folosește predicatul *write:*

write(X).

Acest predicat are ca efect scrierea la terminal a termenului *X*. Dacă *X* nu este inițializat se tipărește numărul variabilei (de exemplu _245). Predicatul *write* este utilizat pentru definirea de alte proceduri pentru realizarea ieșirii formatate. În acest scop se folosesc alte predicate predefinite și anume:

— nl, pentru trecerea cursorului la o nouă linie *(new line)*;
— tab(n), pentru deplasarea cursorului spre dreapta cu *n* spații (nu trebuie confundat cu tasta TAB de la terminal).

Cu aceste elemente se definește predicatul pp(X, Y) *(pretty-print)*, care tipărește într-o formă citibilă componentele listei X.

De exemplu, lista:

[a, b, [c, d, e], f, g]

va fi tipărită sub forma:

```
a
b
    c
    d
    e
f
g
```

adică se decalează cu două coloane spre stînga componentele sublistelor. Predicatul pp(x, y) se definește astfel [1]:

```
pp([H| T], I) : — !, J is I+2, pp(H, J), ppx(T, J), nl.
pp(X, I): — tab (I), write(X), nl.
ppx([ ], _).
ppx([H | T], I) : — pp(H, I), ppx(T, I).
```

Se observă că al doilea argument al predicatului *pp* este un contor ce stabilește coloana de la care începe scrierea. Inițial el este zero(printr-un apel pp(L, 0)).

Prima regulă referitoare la *pp* tratează situația în care primul argument este o listă. În acest caz scrierea începe de la o nouă coloană (prin incrementarea lui J cu 2), tipărindu-se capul de listă pp(H, J).

Scanned with OKEN Scanner

După aceea se tipărește fiecare element din coada listei (T), apelînd predicatul ppx(T, J), cu începere din aceeași coloană ca și capul de listă (coloana J).

A doua regulă referitoare la pp(x, I) permite scrierea unui element X ce nu este o listă. Scrierea se face din coloana I (tab(I)) prin predicatul *write*, după care urmează salt la o nouă linie prin predicatul *nl*. Tăietura '!' îngheață procesul de căutare la prima regulă, pînă la scrierea completă a listei, adică în caz de insucces sistemul PROLOG nu mai continuă parcurgerea bazei de date.

Un alt predicat de scriere este *display*(X), care respectă însă forma de reprezentare internă a argumentului. Diferența dintre *write* și *display* se observă foarte simplu din exemplul următor:

```
? — write(a+b+c), nl, display(a+b+c), nl.
a+b+c
+(a, +(b, c))
```

Citirea termenilor

Citirea de la terminal se face cu ajutorul predicatului *read*:

```
read (X)
```

Variabila X se inițializează cu șirul introdus de la terminal.

Termenul introdus de la terminal trebuie să se termine cu punct(.) și un caracter netipăribil (de exemplu RETURN sau spațiu).

Dacă se testează rezultatul citirii, iar argumentul X este deja inițializat, atunci valoarea introdusă de la terminal se compară cu valoarea inițială, iar succesul operației *read* depinde de concordanța dintre cele două valori.

Fie exemplul următor:

```
răspuns(1, ['Prolog', 'este', 'un', 'limbaj']).
răspuns(2, ['Prolog', 'se', 'învață', 'ușor']).
pr([]) : — nl.
pr([H | T]) : — write(H), tab(1), pr(T).
hello : — pr(['Ce', 'răspuns', 'doriți'?']), read(X),
           răspuns(X, Y), pr(Y).
```

Predicatul *pr* permite tipărirea într-o formă obișnuită limbajului curent a unei liste formată din cuvinte.

Dacă se formulează întrebarea:

```
? — hello.
```

atunci sistemul PROLOG răspunde:

Ce răspuns doriți?

Scanned with OKEN Scanner

și așteaptă introducerea unui șir de la terminal (prin predicatul *read*). Să presupunem că răspundem:

2.

În acest caz sistemul PROLOG tipărește:

Prolog se învață ușor.

Citirea și scrierea de caractere

Entitatea cea mai mică ce se poate citi sau scrie este caracterul. El este tratat ca un întreg, a cărui valoare este codul ASCII. Există mai multe predicate pentru scrierea sau citirea de caractere.

Dacă X este inițializat cu un cod ASCII corespunzător unui caracter tipăribil atunci:

? — put(X).

tipărește caracterul corespunzător codului lui X. De exemplu,

? — put(112), put(114), put(111), put(103), put(114), put(97), put(109).

are ca efect afișarea la terminal a șirului de caractere:

program

Întrebarea:

? — put(97), nl, tab(1), put(98), nl, tab(2), put(99).

are ca efect afișarea formatată:

a
 b
  c

Cu ajutorul predicatului *put* se poate defini un nou predicat pentru tipărirea la terminal a unei constante șir. Exemplu:

```
scrie_șir([]).
scrie_șir([H | T]) : — put(H), scrie_șir(T).
```

Astfel, la întrebarea:

? — scrie_șir („Acesta este un exemplu").

sistemul PROLOG va răspunde:

Acesta este un exemplu.

Caracterele se citesc cu ajutorul predicatului *get*(X) sau *geto*(X). Predicatul *get*(X) ignoră caracterele netipăribile și atribuie lui X primul cod corespunzător unui caracter tipăribil. Predicatul *geto*(X) inițializează pe X cu orice cod ASCII.

Scanned with OKEN Scanner

Predicatele prezentate anterior realizează operațiile de citire/scriere din *fișierul curent de intrare*, respectiv *fișierul curent de ieșire*. În mod normal aceste fișiere sînt reprezentate de terminal. Dar ele se pot utiliza pentru scrierea în fișiere, sau citirea din fișiere aflate pe suporturi magnetice, de exemplu discul.

Specificatorul de fișier apare ca atom și depinde de sistemul de calcul pe care se execută PROLOG. Dacă predicatul *geto*(*X*) sau *read*(*X*) întîlnește sfîrșitul de fișier, atunci argumentul X ia o valoare specifică, de exemplu 26 (control-Z). Există un fișier standard recunoscut de sistemul PROLOG — *user*, ce reprezintă terminalul utilizatorului.

Scrierea în fișierele de pe suporturi magnetice se face cu aceleași predicate prezentate mai înainte, dar trebuie schimbat fișierul curent de ieșire. Acest lucru se realizează cu ajutorul predicatului *tell:*

```
tell(X)
```

unde *X* este un specificator de fișiere (atom). Dacă *X* specifică un fișier ce nu există, atunci sistemul PROLOG creează fișierul respectiv și îl deschide pentru scrierea de date în el. Cînd *X* specifică un fișier ce există deja, atunci sistemul PROLOG deschide fișierul respectiv în adăugare *(append)* (face excepție de la această regulă prima folosire a predicatului *tell* în cadrul programului; în acest caz, dacă fișierul există deja se reinițializează, în sensul că scrierea de date se va face de la începutul lui).

În situația că *X* nu conține un nume de fișier (este neinițializat sau specificatorul de fișier este incorect) se raportează eroare.

Predicatul *telling* se utilizează pentru a obține numele fișierului curent de ieșire.

```
telling(X),
```

unde *X* va conține specificatorul fișierului curent de ieșire.

Predicatul *told* închide fișierul curent de ieșire și are ca efect comutarea lui pe terminal. În cazul în care un fișier curent de ieșire este comutat pe alt fișier fără a fi închis (prin *told*), atunci cînd se revine la el se va face scrierea în continuare *(append)*.

Exemplu:

```
tell(A), write(X), write(Y), tell(B), write(Z), tell(A),
                                                write(W).
```

În fișierul specificat de atomul *A* se vor scrie datele specificate de *X*, *Y*, *W*.

Scanned with OKEN Scanner

Fișierul curent de citire se specifică astfel:

see(X)

unde *X* este un specificator de fișier.

La prima folosire a predicatului *see* în cadrul programului PROLOG se va deschide fișierul *X* la începutul lui. În comutările ulterioare citirea se face din locul unde s-a rămas, pînă se închide explicit cu predicatul *seen*. Fișierul curent de intrare se poate afla cu ajutorul predicatului *seeing*:

seeing(X)

în care *X* va conține specificatorul fișierului curent de intrare.

Inserarea unui text într-un program PROLOG se realizează cu ajutorul predicatului *consult*:

consult(X)

unde *X* este un specificator de fișier al textului (program PROLOG) ce se inserează în programul curent. Trebuie precizat că toate clauzele (reguli, fapte) întîlnite în fișierul specificat se adaugă la sfîrșitul bazei de date a programului.

Dacă se întîlnesc întrebări în fișierul consultat ele se vor prelucra în mod normal, cu deosebirea că nu se tipăresc răspunsurile.

Predicatul *reconsult*(*X*) specifică un fișier *X* ce se inserează în programul curent, cu deosebirea că regulile sau faptele deja specificate în programul curent se înlocuiesc cu cele din fișierul *X*. În general, acest predicat se folosește pentru corectarea programului PROLOG. Corectarea se poate face și direct de la terminal dacă se specifică:

reconsult(user).

În exemplul următor se arată cum se utilizează predicatele *consult* și *reconsult*, cu intrări de la terminal, în scopul punerii la punct a unui program. Se dorește scrierea unui program care să inverseze elementele unei liste. Astfel, lista [a, b, c] devine prin inversare [c, b, a]. Mai întîi se va defini un predicat *adaugă* pentru construirea unei liste. Lucrul de la terminal decurge ca mai jos (comentariile sînt delimitate de perechile de caractere /*și */):

```
adaugă([A | B], C, [A | D]) : — adaugă(A, C, D).
adaugă([ ], X, X).
invers([ ], [ ]).
invers([A | B], C): — invers(B, D), adaugă(D, [A], C).
/* se testează inversarea unei liste */
```


Scanned with OKEN Scanner

```
? — invers([a, b, c, d], X).
nu
/* inversarea nu merge. Se testează dacă adăugarea este corectă */
? — adaugă([a, b, c, d], [e], X).
nu
/* adăugarea nu este corectă. Se corectează predicatul */
? — reconsult(user).
 adaugă([A | B], C, [A | D]) : — adaugă(B, C, D).
ctrl-Z
/* se testează inversarea */
? — invers([a, b, c, d], X).
nu
/* predicatul reconsult a suprimat și predicatul adaugă ([ ], X, X) */
/* se introduce */
? — consult(user).
adaugă([ ], X, X).
ctrl-Z
/* se testează inversarea */
? — invers([a, b, c, d], X).
X = [d, c, b, a]
```

Prin ctrl-Z s-a notat introducerea caracterului CONTROL+Z, prin care se indică sfîrșitul de fișier în intrare.

O facilitate oferită de sistemul PROLOG este și scrierea fișierelor consultate și reconsultate într-o singură listă, în care apare în clar numele fiecărui fișier. Numele fișierului reconsultat este precedat de caracterul '__' sau '$' în alte implementări. De exemplu, întrebarea:

```
? — [fișier1,__fișier2, fișier3].
```

este echivalentă cu secvența:

```
? — consult(fișier1).
? — reconsult(fișier2).
? — consult(fișier3).
```

sau, mai scurt, cu întrebarea:

```
? — consult(fișier1), reconsult(fișier2), consult(fișier3).
```

1.9. Predicate predefinite

Sistemul PROLOG pune la dispoziția utilizatorului o serie de predicate definite în cadrul limbajului, numite și predicate predefinite. Unele dintre ele — *read*, *write*, *put*, *get* etc. — au fost prezentate în paragrafele precedente.

Scanned with OKEN Scanner

Aceste predicate predefinite îl ajută pe utilizator să-şi construiască predicatele programului său într-o manieră simplă şi elegantă. În continuare, se vor prezenta cîteva din cele mai utilizate predicate din PROLOG.

Succes şi insucces

Există două predicate pentru a specifica succesul sau insuccesul execuţiei unei clauze. Acestea sînt *true*, respectiv *fail*:

„true" indică întotdeauna succes;

„fail" indică neîndeplinirea unei clauze, folosindu-se în general împreună cu tăietura '!'.

Exemplu:

```
p :— !, fail.
```

În acest caz, dacă prelucrarea ajunge la *p* se abandonează satisfacerea clauzei.

Predicatul *fail* se mai utilizează atunci cînd se doreşte obţinerea tuturor soluţiilor posibile, forţînd parcurgerea întregii baze de date, fără a introduce ';⟨CR⟩' la fiecare răspuns. De exemplu,

```
lista(1, [a, b, c]).
lista(2,[x, y, z]).
lista(3, [A, B, C]).
```

Prin întrebarea următoare se listează toate listele din baza de date:

```
? — lista(X, Y), fail.
```

Sistemul răspunde:

```
X = 1    Y = [a, b, c]
X = 2    Y = [x, y, z]
X = 3    Y = [A, B, C]
```

Prin adăugarea predicatului *fail* după întrebare se forţează „insucces" ceea ce determină sistemul PROLOG să continue examinarea bazei de date pentru a obţine „succes". În final *fail* forţează răspunsul „nu".

Determinarea tipului

Execuţia unui predicat depinde de tipul argumentelor sale.

O variabilă poate fi iniţializată sau nu. Există două predicate ce specifică acest lucru şi anume:

var(*X*): ia valoarea 'adevărat' dacă *X* este neiniţializat;

nonvar(*X*): ia valoarea 'adevărat' dacă *X* este iniţializat.

Scanned with OKEN Scanner

De exemplu, o regulă se poate abandona dacă o variabilă nu este inițializată:

```
p(X, N) : — var(X) , !, fail.
```

Predicatul *atom*(*X*) ia valoarea 'adevărat' dacă *X* este un atom. *Exemplu*:

```
? — atom(abc).
da
? — atom(124).
nu
? — atom('abcd?').
da
? — atom(„abcd").
nu
? — atom(X).
nu
? — atom([a,[b, c], d]).
nu
```

Predicatul *integer*(*X*) ia valoarea 'adevărat' dacă *X* este un număr întreg, iar *atomic*(*X*) ia valoarea 'adevărat' dacă *X* este un număr întreg sau un atom. El poate fi definit astfel:

```
atomic(X): — atom(X).
atomic(X): — integer(X).
```

Tratarea clauzelor ca termeni

Sistemul PROLOG permite utilizatorului să examineze și să modifice clauzele din programul său. Există predicate predefinite care permit: construirea unei structuri ca o clauză; adăugarea unei clauze bazei de date; eliminarea unei clauze, reprezentată de o structură, din baza de date.

Se observă că o regulă este tratată ca o structură. Într-adevăr, funcția ':—' de definire a regulii poate deveni nume de structură. De exemplu, regula:

```
p(X, Y):— q(X, a)
```

se reprezintă sub formă de structură astfel:

```
':—'(p(X, Y), q(X, a)).
```


Scanned with OKEN Scanner

Dacă regula este compusă, atunci virgula ce desparte predicatele devine și ia nume de structură. De pildă, clauza:

per(X, Z) : — p(X, Z), q(Y, Z)

se transformă în structură astfel:

':—'(per(X, z) ,',(p(X, Z), q(Y, Z))).

În continuare se prezintă predicatele ce permit examinarea și modificarea clauzelor.

Predicatul *listing*(*X*), unde *X* este un atom, permite tipărirea în fișierul curent de ieșire a tuturor predicatelor de nume X; în felul acesta se pot examina și eventual corecta. De exemplu, în programul de inversare liste din § 8.4 presupunem că predicatul „adaugă" din definiția regulii „invers" a fost scris greșit:

```
? — invers([a, b, c, d], X].)
nu
/* inversarea nu a reușit */
/* se listează toate definițiile pentru „invers" */
? — listing (invers).
invers([ ], [ ]).
invers([__28|__43],__19) : —
invers( __43, __35), adaugă(__35, [__28],__19).
```

Ca argumente ale predicatelor *invers* și *adaugă* apar numerele variabilelor.

Prin predicatul *clause* (*X*, *Y*) se caută în baza de date o clauză de nume *X* și corp *Y* și se afișează argumentele sale.

Dacă nu există nici o clauză *X* se anunță „nu". În caz că există mai multe clauze de nume *X* se va afișa prima întîlnită, după care prin ';⟨CR⟩' utilizatorul le poate afișa și pe celelalte.

Referitor la fapt (clauza nu are corp) se atribuie variabilei Y valoarea 'adevărat'.

Exemplu:

```
adaugă ([ ],X, X]).
adaugă ([A | B], C, [A | D]) : — adaugă (B, C, D).
? — clause(adaugă[A, B, C], Y).
A = [ ], B = __28, Y = true;
A = [__28 | __29], B = __30, C = [__28 |__31], Y = adaugă__29,
                    __30,__31);
no
```

Predicatele *asserta*(*X*) și *assertz*(*X*) adaugă o nouă clauză la baza de date. *Asserta*(*X*) adaugă clauza *X* la începutul bazei de date, iar

Scanned with OKEN Scanner

assertz(*X*), clauza *X* la sfîrșitul bazei de date. *X* trebuie să fie inițializată cu o clauză.

Predicatul *retract*(*X*) permite eliminarea clauzei *X* din baza de date.

Se elimină prima clauză *X* din baza de date. Dacă utilizatorul răspunde cu ';⟨CR⟩' se elimină următoarea clauză *X* ș.a.m.d.

Exemplu: se construiește predicatul *retractall* care elimină toate aparițiile unei clauze din baza de date:

```
retractall(X): — retract(X), fail.
retractall(X): — retract((X:—Y)), fail.
retractall(_).
```

Crearea și accesul la componentele unei structuri generale

În PROLOG există trei tipuri de predicate, *functor*, *arg* și *univ*, ce permit descrierea și manipularea structurilor în sensul general.

Sintaxa predicatului *functor* este următoare:

```
functor(T, F, N)
```

unde *T* este o structură sau un atom, iar *F*, *N*, variabile. Dacă *T* este atom atunci *F* ia numele atomului și *N* devine 0. Dacă *T* este o structură, atunci *F* ia numele structurii, iar *N* se inițializează cu numărul de componente al structurii.

Exemplu:

```
? — functor(f(a, b, c), F, N).
F = f,  N = 3
? — functor(a + b, F, N).
F = +,  N = 2
? — functor([a, b, c], F, N).
F = .,  N = 2
? — functor(program, F, N).
F = program,  N = 0
```

În cazul în care *T* nu este inițializat, atunci *F* și *N* trebuie să fie inițializate. În această situație se creează o structură *T* cu numele specificat în *F* și un cu număr de componente precizat în *N*. Elementele structurii vor fi variabile neinițializate.

Ca o aplicație a predicatului *functor* este utilizarea lui la copierea unei structuri. Pornind de la o structură existentă se creează o nouă structură, cu același număr de componente (dar neinițializate). Fie *S* o structură existentă.

Scanned with OKEN Scanner

Predicatul de copiere se scrie astfel:

copiază (S, X) : — functor(S, F, N), functor(X, F, N).

unde X este argumentul pentru noua structură. La primul apel *functor*(*S*, *F*, *N*), *S* fiind o structură, se va iniţializa *F* şi *N* respectiv cu numele structurii şi numărul de argumente. La al doilea apel *functor*(*X*, *F*, *N*) *X* nefiind iniţializat, iar *F* şi *N* fiind iniţializate se creează o structură descrisă de *F* şi *N*.

În final se obţine structura *X* ca o copie a structurii *S*.

Predicatul *arg* are următoarea sintaxă:

arg(N, T, A)

unde *N* este numărul unui argument din structura *T*. Dacă acest argument este acelaşi cu cel specificat în *A*, atunci se indică succes, în caz contrar — eroare. Parametrii *N* şi *T* trebuie să fie iniţializaţi. Dacă *A* nu este iniţializat, atunci se iniţializează cu argumentul numărul *N* din *T*.

Exemplu:

```
? — arg(1, a + (b + c), X).
X = a
? — arg(1, a + b + c, b).
nu
? — arg(2, a(2, a(b, c(d)),X).
X = c(d)
? — arg(2, [a, b, c], X).
X = [b, c]
```

Predicatul = .. pronunţat *univ*, se foloseşte pentru a obţine împreună componentele unei structuri sau a construi o structură, atunci cînd se furnizează lista argumentelor. Notaţia:

X = .. L

se citeşte astfel: *L* este o listă formată din numele *X* şi urmată de argumentele lui *X*. Dacă *X* este iniţializat se creează lista respectivă şi se atribuie lui *L*. În caz contrar lista se va folosi pentru a-i construi o structură (situaţie în care capul de listă *L* trebuie să fie un atom).

Exemplu:

```
? — a(b, c, d) = .. L.
L = [a, b, c, d].
? — [a, b, c, d] = .. L.
L = [., a, [b, c, d]]
```


Scanned with OKEN Scanner

```
? — (a + b) = .. X.
X = [+, a, b].
? — (a + b) = ..[+ ,X, Y].
X = a, Y = b.
? — [a, b, c, d] = ..[X | Y].
X = ., Y = [a, [b, c, d]].
? — X = .. [a, b, c, d].
X = a(b, c, d).
```

Dacă predicatele *functor*, *arg* și *univ* creează și manipulează structuri, predicatul *name* prelucrează atomii. Un atom este un șir de caractere ASCII. Predicatul *name* se folosește pentru a identifica șirul de caractere al unui atom sau pentru a găsi un atom dintr-un șir de caractere. Formatul:

name(A, L)

are următoarea semnificație: caracterele atomului *A* formează lista *L*. Dacă *A* este inițializat se creează lista de caractere și se atribuie lui *L*. În caz contrar se utilizează lista *L* pentru a forma un atom ce se atribuie lui *A*. Exemple:

```
? — name (lista, L).
L = [108, 105, 115, 116, 97]
? — name(X, [108, 105, 115, 116, 97]).
X = lista
? — name(carte, „carte").
da
? — name(program, „prog").
nu
```

Modificarea parcurgerii bazei de date

Există două predicate care afectează modul de examinare a clauzelor din baza de date — !(tăietura) și *repeat*, primul descris în paragrafele precedente.

Predicatul *repeat* asigură generarea mai multor soluții pe baza revenirii înapoi în examinarea bazei de date *(backtracking)*. Fie exemplul următor:

caracter(X): — repeat, geto(X).

Predicatul *repeat* indică repetarea predicatului *geto* de fiecare dată cînd s-a epuizat clauza caracter. Astfel, prin comanda

? — caracter(X)

Scanned with OKEN Scanner

se asigură citirea succesivă a caracterelor din fișierul curent de intrare.

La fel, utilizîndu-se predicatul *repeat*, se poate da o nouă definiție a predicatului *get* (deși acesta este predefinit).

Se știe că predicatul *get* ignoră caracterele netipăribile (cu codul ASCII $\leqslant 32$). Clauza *get* se poate defini astfel:

```
caracter(X) : — repeat, geto(X).
get(X) : — caracter(X), X > 32, !.
```

Dacă un caracter citit din fișierul curent de intrare are codul mai mare decît 32, atunci insuccesul provoacă venirea înapoi pentru alegerea unei soluții, iar predicatul *repeat* forțează reluarea predicatului *geto*. În felul acesta *get* furnizează numai caractere care pot fi tipărite.

Alți operatori și predicate

Operatorul ',' a fost deja prezentat. El reprezintă conjuncția „și", fiind asociativ stînga, adică X, Y, Z are semnificația (X, Y), Z.

Operatorul ';' reprezintă disjuncția („sau"). Clauza X; Y se termină cu succes dacă *X* sau *Y* este îndeplinit.

Predicatul *call* are sintaxa următoare:

```
call(X).
```

unde *X* este o clauză de îndeplinit. *Call*(*X*) se termină cu succes dacă *X* este îndeplinit cu succes. Predicatul *call* este necesar, deoarece la construirea de clauze utilizînd predicatele *functor*, *arg* sau =.. există posibilitatea să se apeleze clauze ce posedă un *functor* care nu este cunoscut în momentul scrierii programului. Rolul predicatului *call* se va observa în exemplele din paragrafele următoare (deși unele versiuni ale limbajului PROLOG nu includ acest predicat).

Predicatul == reprezintă o egalitate mai strictă decît =. Astfel, dacă $X == Y$ este „adevărat", atunci și $X = Y$ este adevărat, dar reciproca nu este întotdeauna adevărată. Predicatul = consideră că o variabilă neinițializată este egală cu orice, pe cînd predicatul == presupune o variabilă neinițializată ca fiind egală numai cu o variabilă neinițializată ce partajează aceeași locație în memorie.

Exemplu:

```
? — X==Y.
nu
? — X==X.
```


Scanned with OKEN Scanner

```
X = - 27
?- X=Y, X==Y.
X = _27, Y = _27
```

Analog cu \= există și predicatul \== care indică succes cînd == indică insucces. El poate fi definit în PROLOG astfel:

```
X\==Y :- X==Y, !, fail.
X\==Y.
```

Referitor la operatorii pentru evaluarea expresiilor aritmetice, se face următoarea observație: variabilele operatorilor +, —, *, mod, <, >, >=, <= trebuie să fie inițializate, altfel se semnalează eroare.

Predicatul *trace* permite urmărirea pas cu pas a îndeplinirii tuturor clauzelor din program. Predicatul *notrace* stopează efectul predicatului *trace*.

În timpul execuției programului, examinarea unor clauze se face cu ajutorul predicatului *spy*. Sintaxa lui este:

```
spy p
```

unde *p* poate fi:

- un atom, situație în care se pune cîte un punct de examinare pentru toate predicatele ce conțin acest atom, indiferent de numărul de argumente ale predicatului. De exemplu, *spy sort* introduce puncte de examinare pentru toate clauzele „sort", indiferent de numărul de argumente;
- o structură cu un argument numeric, de exemplu, *spy sort*(2), în care caz se introduc puncte de examinare numai pentru clauzele „sort" cu două argumente;
- o listă, de pildă *spy sort*(2), *citire*(3); se introduc puncte de examinare pentru toate clauzele „sort" cu două argumente și toate clauzele „citire" cu trei argumente.

Predicatul *debugging* permite tipărirea punctelor de examinare.

Predicatul *nodebug* elimină toate punctele de examinare.

Predicatul *nospy* permite eliminarea selectivă a punctelor de examinare. Sintaxa lui este:

```
nospy P
```

unde *P* are aceeași semnificație ca și la *spy*. De exemplu, *nospy sort*(2), *citire*(3) elimină punctele de examinare pentru toate predicatele „sort" cu două argumente și toate predicatele „citire" cu trei argumente.

Scanned with OKEN Scanner

1.10. Aplicații

Prelucrarea listelor

În cele ce urmează sînt prezentate cîteva predicate de bază ce sînt utilizate în manipularea listelor [1]:

• ultim(X, L): furnizează ultimul element din lista *L*.

```
ultim(X,[X]).
ultim(X, [_ Y]) :—ultim(X, Y).
```

Exemplu:

```
? — ultim(X, [a, b, c, d]).
X = d
```

• consec(X, Y, L): pune la dispoziție valoarea 'adevărat' dacă elementele *X*, *Y* sînt consecutive în lista L.

```
consec(X, Y, [X, Y|_]).
consec(X, Y, [_| Z]) :— consec(X, Y, Z).
```

• adaugă (X, Y, L): construiește lista *L* prin concatenarea listelor *X* și *Y*.

```
adaugă([], L, L]).
adaugă([X | L1], L2, [X | L3]): — adaugă (L1, L2, L3).
```

Terminarea procesului de concatenare are loc atunci cînd primul argument devine „listă vidă". Definiția predicatului este recursivă și se bazează pe următoarele reguli;

— primul element din prima listă (*X*) va fi de asemenea primul din lista finală;

— lista *L2* (argumentul al doilea) se va adăuga cozii primei liste (*L1*) pentru a forma coada (*L3*) a celui de-al treilea argument;

— pe măsură ce se înlătură primul element al primei liste, aceasta se reduce treptat ajungîndu-se la lista vidă, ceea ce constituie condiția de terminare.

• inv (L, M): inversează ordinea elementelor listei *L* și produce lista *M*. Procesul de inversare se realizează prin adăugarea capului de listă la sfîrșitul cozii deja inversate. Procesul de inversare se definește recursiv și se termină atunci cînd primul element devine „lista vidă".

```
inv([], []).
inv([X | Y], L) : — inv(Y, Z), adaugă (Z, [X], L).
```


Scanned with OKEN Scanner

• șterge (X, L, M): elimină elementul X din lista L și produce lista M. Dacă X nu există în lista L atunci acțiunea de „ștergere" eșuează.

```
șterge(_, [], []).
șterge(X, [X | L], L]) : — !.
șterge(X, [Y | L], [Y | M]): —șterge(X, L, M).
```

Clauza șterge (_, [], []) s-a adăugat pentru ca predicatul să nu eșueze dacă elementul al doilea este lista vidă.

• elimină(X, L, M): elimină toate aparițiile elementului X din lista L și produce lista M. Procesul de eliminare se termină atunci cînd argumentul al doilea devine „lista vidă", ceea ce înseamnă că s-a parcurs, prin recurență, întreaga listă L.

```
elimină(_, [], []).
elimină(X, [X | L1], L2): — elimină(X, L1, L2).
elimină(X,[Y | L], [Y | M]) :— elimină(X, L, M).
```

• substituie(X, L, Y, M): înlocuiește toate aparițiile elementului X din lista L cu Y și produce lista M. Prelucrarea este asemănătoare ca și pentru ștergerea elementului X.

Clauzele de mai jos prezintă cele trei cazuri posibile ce apar în procesul prelucrării:

— Lista L este vidă și în acest caz prelucrarea se oprește;
— X este găsit în lista L;
— un alt element diferit de X este găsit în lista L.

```
substituie (_, [],_, []).
substituie(X_,[X | L], Y, [Y | M]) : — !, substituie (X, L, Y, M).
substituie(X, [Z | L], Y, [Z| M]): — substituie (X, L, Y, M).
```

• sublistă(L, M): furnizează valoarea „adevărat" dacă L este o sublistă a listei M, adică toate elementele listei L apar în aceeași ordine și consecutiv în lista M.

De exemplu, la întrebarea:

```
? — sublistă([2, 4, 6], [0, 2, 4, 6, 10, 12])
```

se va răspunde „da". În definiția predicatului se utilizează clauza „prefix" pentru găsirea primului element din lista L în lista M:

```
sublistă([X | L1], [X | L2]) : — prefix(L1, L2), !.
sublistă(L1, [ | L2]): —sublistă(L1, L2).
prefix ([],_).
prefix([X | L1], [X | L2]) : — prefix(L1, L2).
```

Reprezentarea și manipularea mulțimilor

Deoarece mulțimea este o colecție de elemente, ea se va reprezenta în PROLOG printr-o listă. Astfel, mulțimea (a, b, c) se notează prin

Scanned with OKEN Scanner

lista [a, b, c]. Se vor reprezenta prin instrucțiuni PROLOG operațiile de bază asupra mulțimilor:

• membru(X, M): furnizează valoarea „adevărat" dacă X este un element al mulțimii M.

```
membru(X, [X |__]).
membru(X, [__| Y]) : — membru (X, Y).
```

• submulțime(L, M): pune la dispoziție valoarea „adevărat" dacă mulțimea L este o submulțime a mulțimii M.

```
submulțime([X | L1], M) : — membru(X, M), submulțime(L1, M).
submulțime([],M).
```

• disjunct(L, M): furnizează valoarea „adevărat" dacă mulțimile L și M sînt disjuncte (nu au elemente comune).

```
disjunct(L, M) : — not(membru(X, L), membru(X, M)).
```

În definiția predicatului disjunct variabila X este neinițializată, ceea ce înseamnă că la apelul *membru*(X, L) va lua mai întîi valoarea primului element, din lista L, apoi al celui de-al doilea etc.

• intersecție(L, M, N): are ca rezultat mulțimea N ce reprezintă intersecția mulțimilor L și M.

```
intersecție([ ], X, [ ]).
intersecție([X|T], Y, [X|Z]) : —
        membru(X, Y), !, intersecție(T, Y, Z).
intersecție([X|T], Y, Z) : — intersecție(T, Y, Z).
```

• reuniune(L, M, N): are ca rezultat mulțimea N ce reprezintă reuniunea mulțimilor L și M.

```
reuniune([ ], X, X).
reuniune([X|T], Y, Z) : — membru(X, Y), !,
        reuniune(T, Y, Z).
reuniune([X|T], Y, [X|Z]) : — reuniune(T, Y, Z).
```

Sortarea

Sortarea elementelor unei liste se face în funcție de un operator. Astfel, pentru o listă de numere întregi se poate utiliza operatorul „<". Dacă elementele listei sînt atomi se poate aplica predicatul *comp*. În continuare, pentru generalizare, se va considera predicatul *ordonat*(X, Y), care ia valoarea „adevărat" dacă argumentele X și Y sînt într-o anumită ordine, de exemplu „X este mai mic decît Y".

Se vor prezenta programele PROLOG pentru mai multe metode de sortare.

Sortarea prin permutări. Se generează mai multe permutări ale elementelor unei liste și se reține acea permutare ale cărei elemente sînt ordonate (în ordine crescătoare, de exemplu).

Scanned with OKEN Scanner

Predicatul *sortare*(*L*,*M*) sortează lista *L* în lista *M*.

Predicatul *permutare*(*L1*,*L2*) realizează lista *L2* printr-o permutare a elementelor listei *L1*.

Predicatul *sortat*(*L*) verifică dacă elementele listei *L* sînt ordonate.

```
sortare(L,M) :— permutare(L,M),sortat(M),!.
permutare(L,[A|B]) :— adaugă(X,[A|Y],L),
                      adaugă(X,Y,Z).
                      permutare(Z,B).

permutare([_],[]).
sortat(L) :— sortat(0,L).
sortat(_,[]).
sortat(A,[B|C]) :— ordonat(A,B),sortat(B,C).
ordonat(A,B) :— A < B.
```

Sortarea prin inserare. Se inserează elementele listei inițiale într-o nouă listă în poziții definite de o relație de ordine.

Predicatul *sortins*(*L*,*M*) ordonează lista *L* în lista *M* prin metoda inserării:

```
sortins([],[]).
sortins([X|Y],M). :— sortins(Y,Z),inser(X,Z,M).
inser(X,[A|B],[A|C)] :— ordonat(A,X),!,inser(X,B,C).
inser(X,Y,[X|Y]).
ordonat(A,B) :— A < B.
```

Metoda bulelor. Se compară două elemente consecutive și dacă nu sînt în ordine se schimbă locul lor. Acest proces se continuă pînă cînd în listă nu se mai produc modificări.

```
sortb(L,M) :— adaugă(X,A,[B|Y],L),
              ordonat(B,A),
              adaugă(X,B,[A|Y],N),
              sortb(N,M).

sortb(L,L).
adaugă([],L,L).
adaugă([A|B],L,[A|C]) :— adaugă(B,L,C).
```

Quicksort. Este o metodă de sortare elaborată de C.A.R. Hoare. Fie lista *L* formată din capul *X* și coada *Y* (L=[X|Y]).

Se împarte lista *L* în două subliste, *M* și *N*, astfel încît:

— toate elementele din *M* sînt mai mici decît *X*;

— toate elementele din *N* sînt mai mari decît *X*;

— ordinea elementelor din *M* și *N* este aceeași ca din *L*.

Predicatul *split*(*X*,*Y*,*M*,*N*) împarte lista L=[X|Y] în listele *M* și *N*, conform celor de mai sus:

```
split(X,[A|B],[A|C],Z) :— ordonat(A,X),split(X,B,C,Z).
```


Scanned with OKEN Scanner

```
split(X,[A|B],C,[A|Z]) :— ordonat(X,A),split(X,A,C,Z).
split(—,[],[],[]).
```

După găsirea listelor *M* și *N* se adaugă lista *N* listei *M*.
Programul quicksort este următorul:

```
quicksort([X|Y],S) :— split(X,Y,A,B),
                      quicksort(A,T),
                      quicksort(B,V),
                      adaugă(T,[X|V],S).
```

Calculul numerelor prime

Este un exemplu netipic pentru PROLOG deoarece presupune efectuarea de calcule matematice. El scoate în evidență unele posibilități „calculatorii" ale acestui limbaj.

Pentru generarea numerelor prime din intervalul (1,*N*) se construiește un program PROLOG ce aplică metoda lui Eratostene:

— se pun într-o sită toate numerele din intervalul [2,*N*];

— se selectează și se scoate din sită cel mai mic număr și se consideră ca număr prim;

— se elimină din sită toate numerele multiple ultimului număr prim găsit;

— dacă sita nu este vidă se continuă de la pasul al doilea.

Se construiesc mai multe predicate:

— *întregi*(*N*,*M*,*L*): generează lista *L* de numere întregi din intervalul (*N*,*M*);

— *exam*(*L*,*L1*): examinează și scoate unele elemente din lista *L*, producînd lista *L1*;

— *scoate*(*N*,*L*,*L1*): elimină din lista *L* toate numerele multiple numărului *N* și produce lista *L1*;

— *prime*(*N*,*L*): creează lista *L* de numere prime din intervalul (1,*N*).

```
prime(N,L) :— întregi(2,N,LS),exam(LS,L).
întregi(X,Y,[X|Z]) :— X <= Y,!, M is X+1, întregi(M,Y,Z).
întregi(_,_,[]).

exam)[],[]).
exam)[I|L1],[I|L2]) :— scoate(I,L1,L3), exam(L3,L2).

scoate(P,[],[]).
scoate(P,[I|L1],[I|L2]) :— not(o is I mod P),!,
                           scoate(P,L1,L2).
scoate(P,[I|L1],L2) :— o is I mod P,!, scoate(P,L1,L2).
```


Scanned with OKEN Scanner

Transformarea structurilor

De multe ori este necesar să se copieze o structură în alta, după ce s-a efectuat o modificare asupra elementelor structurii inițiale. Elementele se modifică prin aplicarea unui predicat oarecare. De exemplu, *coplist*(*P*,*L*,*M*) aplică predicatul *P* asupra elementelor listei *L*, obținîndu-se lista *M*:

```
coplist(_,[],[]).
coplist(P,[X|L1],[Y|L2]) :- R =.. P[X,Y],
                             call(R),coplist(P,L1,L2).
```

Operatorul =.. (univ) formează o regulă ce este apelată pentru a realiza modificarea elementului *X* în *Y*. Ca o simplificare se poate aplica predicatul *P* asupra elementelor listei *L*, dar fără a forma o nouă listă. De exemplu, predicatul *modlist*(*P*,*L*) modifică lista *L* prin aplicarea predicatului *P*:

```
modlist(_,[]).
modlist(P,[X|L1]) :- R =.. [P,X], call(R),
                     modlist(P,L1).
```

Aplicație la limbajele formale

Din teoria limbajelor se cunoaște că gramatica unui limbaj este o mulțime de reguli care specifică modul de formare a cuvintelor limbajului, gruparea lor pentru a forma fraze și ce fraze sînt admise de limbaj.

Problema de a determina dacă o frază dată aparține limbajului poartă numele de problemă de analiză sintactică, iar programul care rezolvă această problemă se numește procesor de analiză sintactică. În acest paragraf se prezintă modul de realizare în PROLOG a unui procesor de analiză sintactică. Fie de exemplu o gramatică pentru formarea unor fraze în limba română.

```
frază ::= subst_frază,verb_frază
subst_frază ::= articol,substantiv
verb_frază ::= verb,subst_frază
verb_frază ::= verb
articol ::= [un]
substantiv ::= [elev]
substantiv ::= [cerc]
verb ::= [desenează]
```

Cu regulile de mai sus se poate forma fraza: „un elev desenează un cerc". Arborele de analiză sintactică este reprezentat în figura III.5.

Scanned with OKEN Scanner

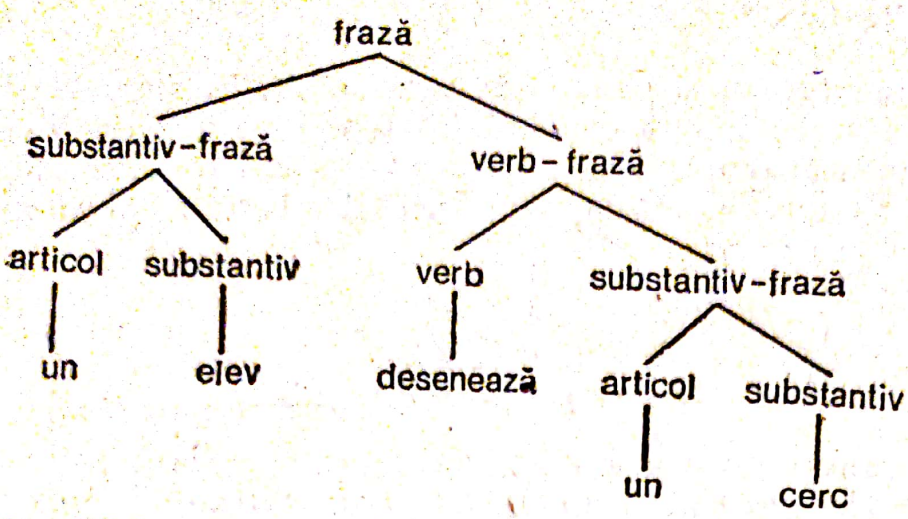


Fig. III.5.

Se intenționează construirea unui program PROLOG care să recunoască frazele ce se pot construi cu regulile de mai sus. De exemplu, să se răspundă cu „da" sau „nu" la întrebarea:

? — frază([un,elev,desenează,un,cerc]).

Se observă că fraza apare sub forma unei liste, iar cuvintele sub forma de atomi. Pentru o aplicație mai complexă se pot introduce cuvintele frazei de la terminal prin utilizarea predicatului *read*.

Pentru scrierea programului PROLOG se observă din regula „frază" că o frază a limbajului este formată dintr-un substantiv_frază și verb_frază. În acest scop se introduc predicatele următoare:

— subst_frază(X): indică dacă X este un element de tip „subst_frază";

— verb_frază (X): indică dacă X este un element de tip „verb_frază". În cazul prezentat o secvență X de cuvinte este o frază dacă se descompune în două secvențe, Y și Z, unde Y este „subst_frază", iar Z este „verb_frază". Deoarece fraza se reprezintă printr-o listă vom utiliza predicatul *adaugă* pentru a pune împreună subsecvențele Y, Z:

```
fraza(X) :— adaugă(Y,Z,X),subst_frază(Y),
            verb_frază(Z).
```

Analog :

```
subst_frază(X) :— adaugă(Y,Z,X),articol(Y),substantiv(Z).
verb_frază(X) :— adaugă(Y,Z,X),verb(Y),subst_frază(Z).
verb_frază(X) :— verb(X).
```

În sfîrșit se pot defini regulile pentru cuvinte:

```
articol([un]).
substantiv([elev]).
```


Scanned with OKEN Scanner

substantiv([cerc]).
verb([desenează]).

Aceasta constituie o primă fază de elaborare a programului PROLOG.

Programul nu corespunde întru totul procesului de analiză a apartenenței frazelor la limbajul definit de gramatică. Fie de exemplu întrebarea:

? —frază([un,elev,desenează,un cerc)].

Variabila X din definiția predicatului „frază" se va inițializa cu lista [un, elev, desenează, un, cerc], dar variabilele Y și Z pot primi ca valori oricare din sublistele listei precedente (cu condiția ca Y și Z să formeze împreună lista X).

Descompunerile posibile ale listei X în sublistele Y și Z sînt următoarele:

Y = [], Z = [un, elev, desenează, un, cerc]
Y = [un], Z = [elev, desenează, un, cerc]
Y = [un, elev] Z = [desenează, un, cerc]
Y = [un, elev, desenează], Z = [un, cerc]
Y = [un elev desenează, un], Z = [cerc].
Y = [un, elev, desenează, un, cerc], Z = [].

Din aceste posibilități se alege o sublistă Y, care satisface definiția predicatului „subst__frază". Dacă sublista Z corespunzătoare nu satisface predicatul „verb__frază", atunci se repetă procesul pentru o nouă sublistă Y, care satisface „subst__frază".

Se pot evita aceste încercări dacă se observă că predicatul *subst__frază* trebuie să aibă doar două cuvinte. Aceasta este o observație particulară valabilă în cazul gramaticii menționate. Astfel, predicatul *subst__frază* se poate defini cu două argumente după cum urmează:

subst__frază(X,Y) :— articol(X,Z),subst(Z,Y)

unde „subst—frază" cere un articol la începutul secvenței X astfel încît restul Z să înceapă cu un substantiv; totodată pentru lista X = [un, elev, desenează, un, cerc] putem avea Z = [elev, desenează, un, cerc] și Y = [desenează, un, cerc].

O convenție similară se aplică referitor la „articol" și „substantiv".

Totuși nu se precizează cîte cuvinte sînt necesare pentru „articol", „subst__frază" sau „frază". „ Această informație se obține pornind de la regulile gramaticii și se exprimă în cuvinte. De exemplu, pentru „articol" se introduce un nou argument:

articol([un|X],X),

ceea ce exprimă faptul că la începutul unei secvențe care începe cu „un" se poate găsi un articol, aceasta separîndu-se de restul secvenței. De ase-

Scanned with OKEN Scanner

menea, se observă că „fraza" este formată din „subst__frază" urmată de „verb__frază".

În final programul PROLOG se scrie astfel:

```
fraza(X,Y) :— subst__frază(X,Z),verb__frază(Z,Y).
subst__frază(X,Y) :— articol(X,Z),substantiv(Z,Y).
verb__frază(X,Y) :— verb(X,Y).
verb__frază(X,Y) :— verb(X,Y),subst__frază(Z,Y).
articol([un|X],X).
substantiv([elev|X],X).
substantiv([cerc|X],X).
verb([desenează|X], X).
```

Versiunea aceasta de program este mai eficientă în ceea ce privește recunoașterea frazelor acceptate de gramatică, dar definiția predicatelor nu mai este totuși așa de clară.

1.11. Depanarea programelor PROLOG

Mai întîi se vor da cîteva indicații privind elaborarea programelor PROLOG, în așa fel încît să se evite introducerea de erori.

În PROLOG termenul de procedură se folosește pentru a desemna colecția de reguli ce definesc un predicat. Se recomandă ca în programul PROLOG procedurile să fie despărțite de o linie-spațiu. În felul acesta se pun mai ușor în evidență regulile de definire ale unui predicat. Dacă o clauză este o conjuncție de mai multe reguli este bine ca fiecare regulă să apară pe o linie separată. De exemplu,

```
permutare([ ],[ ]).
permutare(L,[X|Y]) :—
        adaugă(Z,[X|W],L),
        adaugă(Z,W,T),
        permutare(T,Y).
```

Regulile trebuie însoțite de comentarii pentru a arăta cum se interpretează argumentele unei structuri sau cum se inițializează variabilele. Procedurile nu trebuie să conțină mai mult de 10 reguli; în caz contrar este necesară definirea predicatelor auxiliare.

La scrierea unei proceduri se impune să fie avute în vedere următoarele:

— numele de variabile și de predicate să fie corecte;

— argumentele fiecărui predicat sau *functor* să fie plasate în pozițiile desemnate și să fie în număr corespunzător;

— să se verifice inițializarea variabilelor și să se identifice variabilele ce partajează locații comune;

Scanned with OKEN Scanner

— să se verifice condițiile de terminare în ceea ce privește corectitudinea și plasarea în locurile corespunzătoare;

— să se verifice domeniul fiecărei variabile.

Erorile ce pot apare într-un program PROLOG se împart în două categorii: de sintaxă și, respectiv, de control al fluxului de execuție. Cele mai obișnuite erori de sintaxă sînt următoarele:

- lipsa punctului (.) la sfîrșitul unei clauze. De asemenea punctul trebuie să fie urmat de cel puțin un caracter-spațiu, dacă marchează sfîrșitul unui termen oarecare;
- unele caractere speciale apar în perechi; de exemplu, (,), [,], ghilimele duble("), pentru delimitare șiruri sau ghilimele simple ('), destinate delimitării atomilor. Grupul de caractere /* și */ delimitează un comentariu;
- numele predicatelor sau variabilelor nu se scrie peste tot corect (apar ca nume necunoscute);
- operatorii constituie o altă sursă de erori. Se recomandă utilizarea parantezelor atunci cînd expresiile sînt prea complexe și nu se înțelege prea ușor ordinea de efectuare a operațiilor;
- notațiile folosite în prelucrarea listelor trebuie să se restrîngă la cele prezentate, deoarece nu toate combinațiile sînt corecte. De exemplu, notația [X|Y,Z] nu este corectă (în schimb notația [X,Y|Z] este corectă).

Cele mai frecvente erori de execuție sînt:

- execuție în buclă infinită;
- condiții de terminare insuficiente;
- argumente incorecte pentru un *functor*.

Adesea se întîmplă ca răspunsul obținut să nu fie cel așteptat sau anumite variabile să fie inițializate cu valori nedorite. În acest caz se dorește să se afle în mod rapid cauza erorii și să se corecteze programul. Există două predicate predefinite care permit urmărirea execuției programului (implementarea acestor predicate depinde însă de sistemul PROLOG particular, motiv pentru care este necesar să se consulte documentația în vederea bunei lor utilizări). Aceste predicate sînt *spy* și *trace*. Primul permite introducerea unor puncte de observație în program, iar următorul listează fluxul de control al execuției programului. Eliminarea punctelor de observație se face prin *nospy*, iar eliminarea trasării, prin *notrace*. O altă posibilitate oferită de sistemul PROLOG este depanarea interactivă, adică la fiecare acțiune din program se cere avizul utilizatorului care poate face modificarea fluxului execuției programului.

Scanned with OKEN Scanner

Iată cum arată o sesiune de depanare cu trasare exhaustivă, neinteractivă [1]:

```
? — [user].
adaugă([],X,X).
adaugă([X|Y],Z,[X|W]) :— adaugă(Y,Z,W).
control-Z
? — adaugă([a],[b],X).
CALL adaugă([a],[b],__43)
CALL adaugă([],[b],__103)
EXIT adaugă([],[b],[b]
EXIT adaugă([a],[b],[a,b])
X = [a,b];

REDO adaugă([a],[b],[a,b])
REDO adaugă([],[b],[b])
FAIL adaugă([],[b],__103)
FAIL adaugă([a],[b],__43)
no
```

În succesiunea menționată CALL reprezintă apelul unei clauze, EXIT — o terminare cu succes a unei clauze, FAIL înseamnă clauză nesatisfăcută, iar REDO (reia) reprezintă reluarea unei clauze prin revenirea înapoi (backtracking). Se observă că utilizatorul nu intervine în efectuarea trasării (toate acțiunile întreprinse pentru satisfacerea întrebării utilizatorului se listează).

Dacă depanarea este interactivă, atunci după fiecare acțiune se așteaptă intervenția utilizatorului. De exemplu,

```
? — adaugă([a],[b],X).
CALL adaugă([a],[b],__43)?
```

Execuția programului se oprește după tipărirea semnului întrebării (?), după care urmează ca utilizatorul să introducă o comandă (opțiune). Acestea pot să fie:

— *nodebug* — pentru a elimina toate punctele de observație;
— *notrace* — pentru a elimina trasarea exhaustivă;
— *trace* — pentru a starta trasarea exhaustivă;
— *creep* — pentru a continua trasarea pînă la următoarea clauză de îndeplinit;
— *skip* — se elimină trasarea pînă la următoarea clauză de îndeplinit;
— *leap* — se elimină trasarea pînă la următorul punct de observație;
— *retry* — pentru a relua execuția clauzei curente;
— *or* — pentru a relua clauza curentă cu o nouă alternativă (se utilizează după EXIT);

Scanned with OKEN Scanner

— *fail* — pentru a abandona clauza curentă;

— *break* — se suspendă execuția curentă; se pot introduce noi puncte de observație (cu *spy*). Prin introducerea caracterului control/Z se reia execuția programului;

— *abort* — se forțează teminarea execuției programului;

— *halt* — se oprește execuția programului și se iese din sistemul PROLOG.

Exemplu:

```
? — membru(X,[a,b,c]),membru(X,[b,d,e]).
CALL membru(__44,[a,b,c]) ? creep
EXIT membru(a,[a,b,c]) ? or
REDO membru(a,[a,b,c]) ? creep
CALL membru(__44,[b,c]) ? fail
FAIL membru(__44,[b,c]) ? creep
FAIL membru(__44,[a,b,c]) ? retry
CALL membru(__44,[a,b,c]) ? creep
EXIT membru(a,[a,b,c]) ? creep
CALL membru(a,[d,c,e]) ? fail
FAIL membru(a,[d,c,e]) ? creep
REDO membru(a,[a,b,c]) ? creep
CALL membru(__44,[b,c]) ? creep
EXIT membru(b,[b,c]) ? or
REDO membru(b,[b,c]) ? creep
CALL membru(__44,[c]) ? fail
FAIL membru(__44,[c]) ? retry
CALL membru(__44,[c]) ? creep
EXIT membru(c,[c]) ? creep
EXIT membru(c,[b,c]) ? creep
EXIT membru(c,[a,b,c])? creep
CALL membru(c,[d,c,e]) ? creep
CALL membru(c,[c,e]) ? creep
EXIT membru(c,[c,e]) ? creep
EXIT membru(c,[d,c,e]) ? or
REDO membru(c,[d,c,e]) ? creep
REDO membru(c,[c,e]) ? creep
CALL membru(c,[e]) ? creep
CALL membru(c,[]) ? creep
FAIL membru(c,[]) ? creep
FAIL membru(c,[e]) ? creep
FAIL membru(c,[c,e]) ? retry
CALL membru(c,[c,e]) ? creep
EXIT membru(c,[c,e]) ? creep
EXIT membru(c,[d,c,e]) ? creep
```


Scanned with OKEN Scanner

După depistarea unei erori există două posibilități de corectare a ei: cu ajutorul unui editor de texte (dacă sistemul PROLOG include un astfel de editor); prin comenzile *consult*(*user*) și *reconsult*(*user*), asigurate de sistemul PROLOG.

1.12. Versiuni ale limbajului PROLOG

În paragrafele precedente au fost prezentate noțiunile de bază ale limbajului PROLOG, care se regăsesc, în principiu, în orice versiune de PROLOG. Fiecare sistem de calcul posedă o versiune de PROLOG, ce depinde de tipul de calculator și de sistemul de operare al calculatorului. Implementările limbajului PROLOG diferă în general prin sintaxa regulilor. În specificarea regulilor se folosesc alte semne în locul grupului de caractere ':—'. De asemenea, variabilele apar scrise cu litere mici, iar atomii cu litere mari, sau dacă sistemul de operare nu face distincție între litere mari și mici, atunci variabilele sînt precedate de un caracter special (de exemplu __ sau *). Iată cum apare aceeași regulă în diverse implementări:

```
p(X,Z) :— q(X,Y),r(Y,Z).
p(X,Z) <— Q(X,Y)&R(Y,Z).
P(__X,__Z) :— Q(__X,__Y),R(Y,Z).
+P(*X,*Z) — Q(*X,*Y) —R(*Y,*Z).
((P X Z) ( Q X Y ) ( R Y Z )).
p(x,z) : q(x,y) ; r(y,z).
```

Totodată, apar diferențe de implementare la numărul maxim de caractere admis pentru un atom, numărul maxim de argumente pentru o funcție, numărul maxim de clauze pentru definirea unui predicat etc. Implementările PROLOG pot să difere și prin numărul și felul de predicate predefinite ce sînt puse la dispoziția utilizatorului sau prin facilitățile de depanare ale programului. De asemenea, limbajul PROLOG se poate implementa sub forma unui interpretor sau compilator.

Există versiuni de PROLOG, numite micro-PROLOG, și pentru microcalculatoare (de exemplu. pe Z80 sub sistemul CP/M80, pe 8086/ sub MS/DOS, pe microcalculatoarele Acorn BBC și altele).

Se vor prezenta pe scurt caracteristicile subsistemului PROLOG, versiunea NU7.1[2] implementat sub sistemul UNIX.

Subsistemul PROLOG se lansează în execuție astfel:

```
$ prolog
PROLOG Version NU7.1
?—
```


Scanned with OKEN Scanner

Caracterele '? —' sînt tipărite de PROLOG și ele arată că subsistemul este gata să accepte comenzi.

Consultarea unui fișier se face cu comanda:

? — ['specificator —de —fișier']

unde specificatorul de fișier are următorul format:

[/catalog[/catalog .../]]nume-fișier

În acest caz sistemul PROLOG citește clauzele din fișierul specificat.

Exemplu:

```
? — ['/user/funct/fl']
/usr/funct/fl consulted
yes
```

Prin comanda:

```
? — [user].
```

se indică sistemului PROLOG să citească clauzele de la terminalul de intrare. Aceste clauze se adaugă la sfîrșitul bazei de date deja existentă Pe perioada introducerii acestor clauze nu mai apare prompter-ul ? —.

Sfîrșitul introducerii clauzelor de la terminal se indică prin caracterul special de sfîrșit de fișier, care în general este specific versiunii de sistem UNIX, de exemplu control-d (^d) sau control-z(^z). O dată cu introducerea acestui caracter, sistemul PROLOG arată că este gata pentru o nouă comandă, prin tipărirea grupului de caractere ? —.

Prin caracterul ';' (urmat de „carriage return") se indică continuarea parcurgerii bazei de date pentru obținerea unui nou răspuns la o întrebare pusă anterior.

Dacă la prompter-ul pentru o nouă comandă se răspunde cu caracterul de sfîrșit de fișier (control-d sau control-z), atunci se iese din subsistemul PROLOG și se revine la procesorul de comenzi „Shell":

```
? —  ^d
End of PROLOG execution
$
```

Declararea operatorilor impune anumite restricții. Orice atom poate fi declarat ca un operator . Dacă se fac declarații multiple pentru un atom, întotdeauna ultima declarație este efectivă.

Tipurile de operatori admise sînt:

op x, x op, y op x, x op y

ce exprimă faptul că operatorul poate fi prefix, postfix sau infix (asociativ la stînga sau la dreapta).

Scanned with OKEN Scanner

Deoarece sistemul UNIX rulează în mare parte pe mini și microcalculatoare apar restricții legate și de dimensiunea memoriei ce poate fi adresată de un program.

Astfel de restricții pot fi:

— numărul de atomi dintr-un program să nu depășească 1 024;
— numărul maxim de caractere dintr-un atom este de 253;
— un functor nu poate să aibă mai mult de 15 argumente;
— un număr întreg este cuprins între 0 și 16 383;
— operațiile aritmetice se fac modulo 16 384;

Execuția programului PROLOG se întrerupe cu comanda ^C (control-C). În acest caz există posibilitatea mai multor acțiuni:

— *abort* — se forțează terminarea execuției programului;
— *continue* — se continuă execuția programului;
—*trace* — se continuă execuția programului cu opțiunea de trasare;
— *notrace* — se continuă execuția programului fără opțiunea de trasare;
— *break* — se suspendă execuția programului curent și se poate lansa un alt program (printr-o nouă copie a a interpretorului în memorie). Ieșirea din „break" se face cu caracterul de sfîrșit de fișier (^d);
— *exit* — se iese din sistemul PROLOG.

Sistemul PROLOG exploatează facilitățile oferite de sistemul de operare. Astfel, se pot lansa simultan în execuție mai multe programe ce comunică prin mecanismul numit „pipe". Ieșirea din sistemul PROLOG are loc, de asemenea, dacă se introduce caracterul „quit" (^\).

Se pot executa programe PROLOG cu clauze și întrebări incluse într-un fișier pe disc. Totodată, ieșirea unui program PROLOG se poate face pe disc. Fișierul standard de ieșire, *tty*, specifică terminalul utilizatorului.

Limbajul PROLOG din sistemul UNIX include o bibliotecă cu reguli gramaticale pentru predicatele predefinite.

Limbajul PROLOG a fost implementat și sub alte sisteme de operare, ca de exemplu sub sistemul DECSYSTEM-10 [3] sau RT-11 [4]. Implementarea sub RT-11 este asemănătoare celei subsistemul UNIX, fiind realizată de aceeași specialiști.

1.13. Concluzie

Exemplele prezentate în acest capitol sînt uneori extrem de simple. Există totuși programe complexe din domeniul inteligenței artificiale care au fost scrise în întregime în PROLOG. Au fost elaborate programe PROLOG pentru aplicații industriale, reacții chimice, arhitectură etc.

Scanned with OKEN Scanner

și s-au făcut investigații pentru posibilitatea utilizării limbajului PROLOG în planificarea producției și controlul proceselor industriale.

Ca limbaj, PROLOG este foarte eficient în sistemele-expert [8] pentru exploatarea bazei de date și interfața cu utilizatorul. Dintre sistemele-expert realizate în PROLOG menționăm Chat [9], care răspunde la întrebări relative la geografia continentelor, și ORBI [10], pentru evaluarea resurselor mediului natural.

PROLOG poate fi aplicat în multe ramuri de cercetare; în prezent este asociat tot mai mult cu problemele de inteligență artificială, dovedindu-se un limbaj practic și eficient. Viitorul ne va rezerva în mod sigur o dezvoltare a acestui limbaj, cu apariția de noi versiuni specifice pe domeniu, precum și utilizarea lui în tot mai multe sectoare de activitate.

CAPITOLUL 2

MICRO-PROLOG

2.1. Observații generale

Micro-PROLOG — funcțional astăzi pe microcalculatoarele Z80 sub sistemul CP/M80, pe microcalculatoarele 8088/86 sub sistemul MSDOS, pe microcalculatoarele Acorn BBC și pe alte calculatoare ce lucrează sub sistemul de operare UNIX — este un derivat al limbajului PROLOG, diferențele fiind de natură sintaxică în formatul clauzelor sau introducerea întrebărilor.

Limbajul micro-PROLOG descris aici a fost implementat pe microcalculatoarele ZX SPECTRUM [6].

Formatul relațiilor unare este postfix:

nume-obiect proprietate

De exemplu, faptul că Ionescu este student se scrie în micro-PROLOG astfel:

ionescu student.

Relațiile binare se scriu în general în formatul infix:

nume-obiect relație nume-obiect.

De pildă, faptul că Radu este prieten cu Vlad se exprimă astfel:

radu prieten vlad.

Formatul prefix este general și se poate aplica la orice relație: unară, binară, ternară etc. Astfel, relația de mai sus se poate scrie:

prieten(radu,vlad).

Scanned with OKEN Scanner

Limbajul permite folosirea celor trei formate ale relațiilor, cu remarca următoare: relațiile unare vor fi afișate de sistemul micro-PROLOG în format postfix, iar cele binare, în format infix, chiar dacă au fost introduse în format prefix.

Caracterul separator dintre argumente este caracterul spațiu. Orice comandă introdusă de la display se termină cu ENTER sau RETURN. Parantezele rotunde delimitează începutul sau sfîrșitul unui argument și deci nu este nevoie să fie urmate sau precedate de „spațiu".

În micro-PROLOG, caracterul '_' de formare a cuvintelor compuse cu înțeles pentru programator este înlocuit cu '-' (caracterul de despărțire în silabe).

Numele de relații sau argumente pot să înceapă cu literă mare: litera mare nu mai are rolul de desemnare a variabilei.

Singura restricție o constituie faptul că un nume de relație sau argument nu poate să înceapă cu o cifră.

De exemplu, 4abc nu este permis ca nume (de fapt el este interpretat de micro-PROLOG ca 4 abc, deoarece o modificare în tipul caracterelor se consideră ca separator).

Caracterul prompter al sistemului micro-PROLOG este &, urmat de caracterul punct (.) și indică faptul că sistemul este gata să primească comenzi de la terminal.

Introducerea clauzelor, de la terminal în baza de date a programului PROLOG, se realizează cu comanda *add*.

Exemplu:

```
&.add (radu prieten vlad)
```

Se observă că argumentul comenzii *add* (clauza) se închide între paranteze rotunde.

Dacă relația de introdus depășește lungimea liniei terminalului, atunci se introduce ENTER înainte de a se ajunge la sfîrșitul liniei, iar pentru linia de continuare sistemul PROLOG va tipări:

```
n.
```

unde *n* este numărul de paranteze „dreapta" ce trebuie introduse în linia de continuare pentru a închide corect parantezele „stînga".

Exemplu:

```
&.add (radu este — prieten — cu <ENTER>
1.vlad)
```

Introducerea relațiilor în format prefix este mult facilitată prin comanda *accept*. Formatul ei este:

```
&.accept nume-relație
```


Scanned with OKEN Scanner

Sistemul micro-PROLOG răspunde cu *nume-relație* urmat de punct (.) și așteaptă introducerea (între paranteze) a argumentelor relației:

```
nume-relație.(arg1 arg2 arg3...)
nume-relație.(arg1 arg2 arg3...)
nume-relație.end
```

Sfîrșitul introducerii de date se semnalează prin cuvîntul cheie *end*.

De exemplu, se dorește introducerea în baza de date a următoarelor relații:

```
student(ion)
student(dan)
```

Cu comanda *accept* se procedează astfel:

```
&.accept student
student.(ion)
student.(dan)
student.end
&.
```

Șirul de caractere „student." este tipărit de către sistemul micro-PROLOG.

Comanda *list* se folosește pentru listarea bazei de date în întregime (cu opțiunea *all*) sau a unei relații specificate:

```
&.list all
&.list student
```

Salvarea bazei de date pe casetă magnetică se face cu comanda *save*:

```
&.save nume-volum
```

unde *nume-volum* este numele bazei de date copiată pe suportul magnetic.

Numele de volum trebuie să fie diferit de orice nume de comandă sau de orice nume de relație din baza de date. Altfel se semnalează eroare.

Restaurarea bazei de date de pe suportul magnetic se realizează cu comanda *load*:

```
&.load nume-volum
```

unde *nume* este eticheta de volum specificată anterior.

Scanned with OKEN Scanner

2.2. Comenzi pentru modificarea bazei de date

Pentru eliminarea unei relații din baza de date se folosește comanda *delete*:

&.delete(relație)

relația care se șterge este specificată între paranteze.

Eliminarea unei relații se poate realiza și prin specificarea numărului ei în baza de date:

&.delete nume-relație n

în care relația specificată prin „nume-relație" se află în poziția *n* în lista tuturor relațiilor din baza de date.

De asemenea, există posibilitatea de a înlocui o relație din baza de date cu alta prin comanda *add*. În acest caz în comanda *add* se specifică și numărul poziției relației din baza de date:

&.add n (relație)

Noua relație specificată prin argumentul „relație" va înlocui în baza de date relația ce ocupă poziția *n*.

Evident că modificarea bazei de date se poate realiza și cu un editor de texte, dacă acesta există în configurația sistemului de operare (vezi editorul de texte al sistemului).

Micro-PROLOG conține mai multe nume de relații numite relații (predicate) predefinite. Numele lor este rezervat și nu poate fi alterat de către programator. De exemplu, dacă se încearcă redefinirea relației *add* sistemul micro-PROLOG răspunde cu un mesaj, în conformitate cu care relația menționată nu poate fi modificată.

Ștergerea tuturor aparițiilor unei relații din baza de date se realizează cu comanda *kill*:

&.kill nume-relație

Prin comandă:

&.kill all

se șterg toate relațiile din baza de date. Această comandă se utilizează după salvarea programului PROLOG pe suport magnetic, pentru a face loc în spațiul de lucru altui program PROLOG.

Reîncărcarea sistemului micro-PROLOG se realizează cu comanda NEW:

&.NEW.

Ca și la comanda *kill all* trebuie salvat programul PROLOG înainte de execuția acestei comenzi. După execuția comenzii respective este

Scanned with OKEN Scanner

necesară reîncărcarea preprocesorul SIMPLE (cu comanda LOAD, ce convertește comenzile micro-PROLOG SPECTRUM în comenzi PROLOG standard):

LOAD SIMPLE.

Caracterul punct (.) de la sfîrșitul comenzii NEW este necesar deoarece orice comandă micro-PROLOG trebuie să aibă cel puțin un argument. În loc de punct se poate scrie orice șir de caractere, de exemplu:

&.NEW hello

2.3. Întrebări

Forma cea mai simplă de întrebare este cea care solicită confirmarea unui fapt. Formatul ei este următorul:

&.*is* (relație1[*and* relație2...])*

unde relație1, relație2... desemnează fapte. Sistemul micro-PROLOG răspunde cu „yes" dacă faptele respective există în baza de date și cu „no" în caz contrar.

Fie baza de date:

```
george prieten vlad
dan prieten vlad
radu prieten ion
ion student
maria student
```

La următoarele întrebări se precizează răspunsurile respective:

```
&.is(radu prieten ion)
YES
&.is(radu prieten vlad)
NO
&.is(x prieten vlad)
YES
```

Prin ultima întrebare încercăm să aflăm dacă „VLAD" are vreun prieten, iar sistemul micro-PROLOG răspunde afirmativ.

Pentru a afla ce prieten are Vlad trebuie pusă întrebarea într-un alt format, și anume cel de obținere a informațiilor din baza de date:

```
&.which(var1[var2...]: întrebare[and                    1
                        întrebare2...])
```

* Parantezele drepte [,] desemnează parametrii opționali.

Scanned with OKEN Scanner

var1, var2, ... sînt nume de variabile, iar întrebare1, întrebare2..., sînt întrebări puse sistemului micro-PROLOG.

Fie de exemplu întrebarea:

&.which(x: x prieten vlad)

Sistemul micro-PROLOG răspunde:

george
dan
No(more) answers

Deci, se afișează toate valorile variabilei *x*, care înlocuite în întrebarea pusă corespund la fapte ce există în baza de date, în continuare afișîndu-se mesajul „no(more) answers", ce arată că s-au epuizat toate răspunsurile posibile.

Variabilele în micro-PROLOG se desemnează prin literele:

x, *y*, *z*, *X*, *Y*, Z.

De asemenea, oricare din literele de mai sus, urmată de o cifră sau un șir de cifre zecimale, este o variabilă.

Astfel, x1, y21, z2, y123 sînt variabile.

Dar x02 și x2 desemnează aceeași variabilă, deoarece 02 și 2 specifică același număr zecimal (2).

O comandă poate conține mai multe întrebări legate prin conjuncția *and* (și). De exemplu, se dorește să se cunoască prietenul comun al lui „george" și „dan":

&.wich(x:george prieten x and
1.dan prieten x)

Se observă că la continuarea comenzii pe linia următoare sistemul micro-PROLOG a răspuns cu „1.", ceea ce înseamnă că așteaptă închiderea comenzii cu o paranteză „dreapta". La întrebarea de mai sus sistemul micro-PROLOG răspunde:

vlad
No (more) answers

În loc de *and* se poate folosi de asemenea caracterul „&".

Exemplu:

&.which(x:radu prieten x & x stundent)
ion
No (more) answers

Dacă se dorește un singur răspuns la o întrebare, atunci trebuie folosit formatul *one*:

&.one(*var*1[var2...]: întrebare1 [*and* întrebare2...])

Scanned with OKEN Scanner

După furnizarea răspunsului sistemul micro-PROLOG cere confirmarea programatorului dacă să continue examinarea bazei de date pentru furnizarea unui nou răspuns.

Exemplu:

```
&.one(x: x prieten vlad)
george
more? (y/n)y
dan
more? (y/n)n
&.
```

Pentru a afla dacă o relaţic există în baza de date se foloseşte întrebarea:

&.*is*(nume-relaţie *defined*)

Sistemul micro-PROLOG răspunde cu YES sau NO, după cum relaţia se află sau nu în baza de date a programului.

Listarea tuturor relaţiilor din baza de date se realizează cu comanda:

```
&.all(x: x dict)
```

Ea este echivalentă cu comanda:

```
&.list all
```

2.4. Relaţii aritmetice

Micro-PROLOG nu este adecvat pentru calcule numerice; totuşi, se pot construi expresii aritmetice cu ajutorul a patru predicate predefinite; SUM, TIMES, LESS şi INT. Aceste relaţii aritmetice se utilizează ca şi celelalte relaţii din baza de date a programului. Predicatul SUM calculează suma sau diferenţa a două numere:

SUM(x y z) calculează $z = x + y$

Exemple:

```
&.is(SUM(10 20 30))
YES
&.which(x: SUM(3.6 —1.5 x))
2.1
No (more) answers
&.which(X: SUM(X 24 134))
110
No (more) answers
```

Predicatul SUM, în momentul evaluării, nu trebuie să aibă mai mult de un argument necunoscut (variabilă). Argumentele pot fi nu-

Scanned with OKEN Scanner

mere întregi sau reale. Dacă nu sînt de același tip, sistemul micro-PROLOG convertește numărul întreg în real. În convenția micro-PROLOG numerele reale conțin punctul zecimal sau litera „E" pentru exprimarea exponentului lui 10.

Exemplu:

12.123 4.1E1 0.05E3 4.0E-3

Numerele pozitive nu trebuie să fie precedate de semnul „+" (altfel se semnalează eroare).

Numerele negative sînt precedate de semnul „—".

Exemple:

—4 —2.5E-3 —27.105E5

Numerele întregi sînt cuprinse între —32767 și 32767.

Numerele reale pot să aibă pînă la 8 cifre semnificative. Exponentul poate avea valori întregi între —127 și 127. Formatul standard al numerelor reale — afișat de micro-PROLOG — conține o singură cifră pentru partea întreagă. De exemplu, dacă se introduce de la terminal 31.35, sistemul micro-PROLOG afișează 3.135E1.

Predicatul INT se utilizează în două moduri:

• ca test pentru a afla dacă un număr este întreg (sau real, dar cu parte fracționară nulă). Exemple:

```
&.is(30 INT)
YES
&.is(2.4 INT)
No
```

• ca relație binară pentru obținerea părții întregi a unui număr real.

Exemple:

```
&.which(x: 67.2 INT x)
67
No (more) answers
&.which(X: —23.5E2 INT X)
—2350
NO (more) answers
```

În cazul utilizării ca relație binară, argumentul variabilă trebuie specificat în poziția a doua.

Predicatul TIMES calculează înmulțirea sau împărțirea a două numere, ori verifică rezultatul acestor două operații:

TIMES (x y z) calculează z = x*y.

Scanned with OKEN Scanner

Exemple:

```
&.is (TIMES 2..5 10))
YES
&.which(x: TIMES(10 2.1x))
2.1E1
No (more) answers
&.which(x: TIMES(x 5 60))
12
No (more) answers
```

Ca și predicatul SUM, predicatul TIMES admite cel mult un argument variabilă (în momentul evaluării).

Predicatul LESS se folosește numai pentru a verifica ordonarea a două numere sau două șiruri de caractere: LESS(x y) produce răspunsul „YES" dacă „x este mai mic ca y". Cînd x, y sînt numere se compară valorile lor. Dacă x, y sînt șiruri de caractere se compară (caracter cu caracter) codul ASCII al caracterelor.

Exemple:

```
&.is. (5 LESS 6)
YES
&.is (ABC LESS ABCDE)
YES
&.is (CARTE LESS CAIET)
NO
```

Cu privire la compararea șirurilor se fac următoarele observații:

— codul ASCII al literelor mari este mai mic decît cel al literelor mici;

— codul ASCII al cifrelor este mai mic decît cel al literelor;

— codul ASCII al caracterului „—" este mai mic decît al cifrelor (și deci al literelor).

Astfel,

```
ABC este mai mic decît abc
ABC1 este mai mic decît ABCD
AB-D este mai mic decît ABCD
```

Evaluarea predicatelor dintr-o expresie complexă cu conjuncții se face de la stînga la dreapta. De pildă, întrebarea următoare este validă, deoarece în momentul evaluării predicatului SUM variabila *y* este definită:

```
&.which(x: TIMES (7 3 y) and SUM(y 10 x))
31
No (more) answers
```


Scanned with OKEN Scanner

2.5. Reguli

În micro-PROLOG o propoziție condițională se definește astfel:

relație if relație1 [and relație2...]

unde *if* se traduce prin „dacă". Propoziția condițională se mai numește și implicație. Relația din partea stîngă lui *if* este denumită concluzie, iar relațiile din partea dreaptă — condiții.

În micro-PROLOG regula se definește ca o propoziție condițională sau relație care conține variabile.

Regulile sînt necesare pentru definiții recursive. Orice definiție recursivă trebuie să aibă cel puțin o regulă nerecursivă, altfel în timpul execuției sistemul micro-PROLOG se rulează pînă cînd epuizează spațiul de lucru. Ca exemplu de regulă recursivă vom defini calculul factorialului unui număr întreg pozitiv, $n! = 1.2.3 \ldots (n-1)n$.

Fie relația: x factorial y care calculează în y factorialul numărului x. Programul de calcul al factorialului unui număr întreg pozitiv este următorul:

```
1 factorial 1
x factorial y if
        1 LESS x and
        SUM(x1 1 x) and
        x1 factorial y1 and
        TIMES(x y1 y)
```

Exemplu de utilizare:

```
&.which(x: 5 factorial x)
120
&.is(4 factorial 24)
YES
```

2.6. Lista

De multe ori este necesar să existe un fapt ce se referă la o listă de elemente. De exemplu, în limbajul natural se spune:

Iarna este zăpadă, ceață, frig.

Acest fapt se poate reprezenta în micro-PROLOG prin mai multe relații:

```
iarna este zăpadă
iarna este ceață
iarna este frig
```


Scanned with OKEN Scanner

Există posibilitatea de a reprezenta relațiile de mai sus printr-o singură relație:

iarna este (zăpadă, ceață, frig)

în care expresia dintre paranteze se numește *listă*.

La întrebarea următoare:

&.which (x: iarna este x)

se va obține răspunsul:

(zăpadă, ceață, frig)
No (more) answers.

Lista are avantajul de a manipula mai multe elemente decît una singură, de unde rezultă o compactare în reprezentarea informației. Dezavantajul ei constă în faptul că prelucrarea este mai laborioasă pentru a avea acces la o entitate din cadrul listei. De exemplu, la întrebarea:

&.is (iarna este zăpadă)

se va răspunde „no" pentru că (în cazul reprezentării prin listă) nu există nici o relație în baza de date conformă cu întrebarea.

O listă de dimensiune oarecare este de forma următoare:

(x1 x2 ... xn).

Elementul x1 se numerte *capul* listei, iar sublista (x2...xn) — *coada* listei. Dacă o listă conține un singur element ea are coada vidă, notată (). Cu privire la apartenența unui element la o listă se poate enunța următoarea regulă: un element face parte dintr-o listă dacă este cap de listă sau aparține cozii listei. Separatorul elementelor listei este caracterul „spațiu".

Capul de listă și coada se specifică utilizîndu-se operatorul | (bară verticală). Expresia (x | y) reprezintă o listă formată din elementul x (cap de listă) urmat de lista y (coada listei). Operatorul | se traduce prin „urmat de".

De exemplu, pentru lista (A B C D) avem x = A și y = (B C D).

Capul de listă poate fi format din mai multe elemente sau poate fi chiar o listă. În acest sens,

(x y | Z) — desemnează o listă formată din elementele x y urmate de o listă altă Z. Exemplu: (a b c d), unde x = a, y = b, Z = (c d);

((x y) | Z) — desemnează o listă ce începe cu o sublistă formată din două elemente și urmată de o listă Z. Exemplu: ((a b) c), unde x = a, y = b, Z = (c);

((x | y) | Z) — desemnează o listă ce începe cu o sublistă formată din cel puțin un element (x). Exemplu: ((a b) c), unde x = a, y = (b),

Scanned with OKEN Scanner

Z = (c).

Fie predicatul „aparține", ce calculează dacă un element face parte dintr-o listă. Definiția lui în micro-PROLOG este:

```
X aparține (X | Z)
X aparține (X | Z) if X aparține Z
```

Exemplu de utilizare:

```
&.all(x : x aparține (A B C D))
```

Răspunsul este:

```
A
B
C
D
No (more) answers
```

Definiția predicatului este recursivă. Răspunsul tipărit este capul de listă, respectiv al listelor (A B C D), (B C D),(C D), (D).

Pentru calculul numărului de elemente al unei liste se definește predicatul „număr-elem" astfel:

```
() număr-elem o
(x | X) număr-elem z if X număr-elem y
        and SUM(y 1 z)
```

Deci, numărul de elemente dintr-o listă este egal cu numărul de elemente din coada listei plus 1 (capul de listă). Exemplu de utilizare:

```
&.which (x : (A B C D) număr-elem x)
4
No (more) answers
```

Predicatul „lungime" caută o listă cu un număr dat de elemente:

```
o lungime ()
y lungime (x|X) if
        y INT aud o LESS y and
        SUM(z 1 y) and z lungime X
```

Dacă numărul *y* specificat este zero se furnizează lista vidă (.) În acest mod se verifică dacă *y* este întreg și pozitiv, apoi se scade 1 obținîndu-se *z*, care trebuie să fie lungimea cozii *X* a listei date.

Predicatul predefinit *isall* furnizează toate soluțiile ca rezultat al unei întrebări sub forma unei liste, dar în ordinea inversă listării cu alte predicate. De exemplu, dacă la întrebarea:

```
&.all (x: x prieten dan)
```


Scanned with OKEN Scanner

se obțin răspunsurile:

george
ion
vlad

atunci la întrebarea:

&.which(x: x isall(y: y prieten dan))

se va obține răspunsul sub formă de listă (în ordine inversă):

(vlad, ion, george)

2.7. Predicate predefinite

În acest paragraf se prezintă formate, exemple și restricții pentru mai multe predicate predefinite în micro-PROLOG:

Negație: **not**
Format: not C, unde *C* este o relație sau o conjuncție de relații între paranteze.

Exemple:

&.which(x: x bărbat & not (x tată y & y bărbat)

Se tipăresc toți bărbații care nu au fii.

&.which (x: x oraș Romania & x populație y & not y LESS 20000)

Se tipăresc orașele din Romania cu o populație mai mare de 200 000.)

Restricție: *not* se aplică la variabile inițializate.

Isall

Format: L isall (A: C)
L este o variabilă sau o listă.

A: C are formatul întrebării *which*, adică variabilă răspuns (A) și relație (C).

Semantica: *L* este o listă a tuturor *A* în ordinea inversă în care sînt întîlnite, astfel că există *C* pentru y1, ..., yk, unde y1', ..., yk sînt variabile locale din *C*.

Restricție: variabilele trebuie să fie inițializate.

Scanned with OKEN Scanner

Se utilizează în special în regim de generare, adică de tipărire a unui răspuns. Dacă *L* este o listă furnizată de programator, atunci se spune că *isall* se folosește în regim de verificare. Acest regim nu se recomandă, mai ales dacă lista are mai mult de un element. De exemplu:

&.is ((Vlad Gelu Dan) isall (x: Sanda prietenă x))

Deși în baza de date, Vlad, Gelu, Dan pot figura ca prieteni ai Sandei, totuși răspunsul la întrebare poate fi nu, deoarece ordinea în care se construiește lista ca răspuns de către *isall* nu conține elementele în aceeași ordine ca cea dată de programator.

Întrebările următoare sînt însă valide în sensul că răspunsul obținut este corect:

&.is (() isall (x: Dan prieten x)
&.is ((Gelu) isall (x: Vlad prieten x)

Prima întrebare verifică dacă Dan nu are nici un prieten, iar cea de-a doua verifică dacă Gelu este singurul prieten al lui Vlad.

Întrebarea următoare:

&.which (x: 3 număr-elem x & x isall (y: Sanda prietenă y))

verifică dacă Sanda are numai trei prieteni, și în caz afirmativ sistemul micro-PROLOG listează numele lor.

Forall

Format: (forall C then C')

unde, *C*, *C'* sînt relații sau conjuncții de relații. Parantezele sînt obligatorii (pentru a desemna sfîrșitul predicatului forall).

Semantica: pentru toate variabilele locale x1, x2 ... xn din *C* și *C'* dacă *C* este îndeplinit, atunci se execută și *C'*.

Restricții: variabilele trebuie să fie inițializate, altfel sistemul micro-PROLOG furnizează date eronate. Forall este echivalent cu dubla condiție și anume:

(forall C then C') <=> not (C and not (C')).

Exemple:

Regula următoare se utilizează pentru a verifica dacă lista *X* este o sublistă a listei *Y*:

X subset Y if
(forall x aparține X then x aparține Y).

Scanned with OKEN Scanner

Pentru a afla dacă două liste au aceleași elemente se definește și predicatul următor:

X identic Y if X subset Y & Y subset X.

Cu aceste predicate se poate verifica dacă Vlad, Gelu și Dan (din exemplul de la prezentarea predicatului *isall*) sînt prietenii Sandei; în acest scop se pune următoarea întrebare:

&.is(x isall (y: Sanda prieten y) &(x identic (Vlad Gelu Dan))

„Identic" nu înseamnă însă că elementcle celor două liste de comparat sînt în aceeași ordine.

Pentru a verifica dacă elementele unei liste sînt în ordine crescătoare se folosește următorul predicat:

X ordonat if (forall (x y) adiacent X then x mmse y), unde predicatele „adiacent" și „mmse" (mai mic sau egal) se definesc astfel:

(x y) adiacent (x y | X)
(x y) adiacent (x | X) if (x y) adiacent X
x mmse x
x mmse y if x LESS y

or

Format: (either C or C')
undc C și C' sînt relații sau conjuncții de relații. Se citește (sau C ori C').
Parantezele sînt necesare. Termenii C, C' pot fi fapte, relații negate, predicate *isall*, *forall* sau condiții *or*.

Exemplu:

Presupunem că avem în baza de date relațiile:

x părinte y if x tată y.
x părinte y if x mamă y.

Acestea se pot exprima printr-o singură regulă, folosind condiția *or*:

x părinte y if (either x tată y or x mamă y).

Cu condiția menționată, predicatul „aparține" se poate defini astfel:

x aparține (y | z) if (either x EQ y or x aparține z)

Expresii: formal o expresie poate fi o constantă; un număr o variabilă, o expresie aritmetică, un apel de funcție, o listă de expresii.

Scanned with OKEN Scanner

Expresiile aritmetice și apelurile de funcții sînt liste speciale.

Format: (expresie operator expresie) unde „operator" poate fi:

* — înmulțire
% sau / — împărțire
\+ — adunare
— sau ~ — scădere

Parantezele sînt obligatorii. Expresia apare deci ca o listă în care argumentul al doilea este un operator. Precedența operatorilor este următoarea: *, % (sau /) sînt evaluate înainte de +, — (sau ~). Operatorii + și — (sau ~) au aceeași prioritate, dar mai mică decît a operatorilor de înmulțire și împărțirea. Evaluarea a doi operatori adiacenți de aceeași prioritate se face în ordinea întîlnirii lor de la stînga la dreapta. Semnul % este consacrat pentru operația de împărțire (dacă folosiți în expresie semnul /, la listare se va utiliza %). La fel semnul — este consacrat pentru operația de scădere. Semnul ~ este util pentru a desemna operația de scădere, atunci cînd nu se spațiază operatorul De pildă,

x — 4 este interpretat ca variabila „x — 4" dar
x ~ 4 este interpretat ca o diferență x — 4.

Exemple:

(x * y + 3/z) (echivalent cu (x * y) + (3%z)).
(x + y / (2 + z)) (echivalent cu (x + (y%(2 + z))).

2.8. Expresii de apel de funcții

Apelurile de funcții permit relațiilor să fie utilizate ca funcții în expresii.

Fie de exemplu definiția relației „împărțire":
împărțire (x y z) if TIMES(y z1 x) & INT(z1 z),

unde INT, un predicat predefinit, se folosește pentru a testa dacă un număr este întreg sau pentru a găsi partea întreagă a unui număr (după cum este folosit aici). Predicatul „împărțire" calculează cîtul z al împărțirii lui *x* cu *y*. Pentru a utiliza relația „împărțire" în expresii trebuie să se declare că un argument depinde de alte argumente și acest lucru se realizează cu comanda *function*:

function împărțire.

Prin această comandă se înregistrează relația „împărțire" ca o funcție specială, ce poate fi utilizată în expresii, ca de exemplu:

x = (împărțire z 6).

Apelul funcției conține doar argumentele necesare pentru a efectua calculele, iar valoarea calculată în argumentul destinație se atribuie unei variabile oarecare (aici x).

Sintaxa apelului de funcție este:

(R E1 E2 ... En — 1),

unde E1, E2, ..., En —1 sînt expresii, iar R este numele unei relații n-are, ce a fost declarată ca funcție prin comanda:

function R.

Expresiile E1, E2, ..., En —1 sînt de fapt argumentele unei relații de forma:

R(V1 ... Vn — 1 x),

în care V1, ..., Vn — 1 sînt valorile expresiilor E1, ..., En — 1.

Valoarea funcției este valoarea ce trebuie atribuită variabilei x. În general, la analiza unei expresii, procesorul de analiză avertizează pe programator dacă o relație este apelată ca funcție, dar nu a fost declarată ca funcție. Lista tuturor relațiilor care au fost declarate ca funcții se obține prin una din comenzile:

```
&.which(x : x func)
&.list func
```

Apclurile de funcții pot să apară ca argumente ale operatorilor aritmetici; de asemenea, expresiile aritmetice pot să fie utilizate ca argumente în apelurile de funcții.

2.9. Condiția de egalitate

Este o formă specială de condiție în care se pot evalua și compara expresii. Ea se exprimă prin semnul „=".

Prin lipsă se consideră că argumentele unei relații nu sînt expresii ci termeni. Semnalarea expresiilor în locul argumentelor se face prin semnul „#", ce se plasează între numele relației și argumente. Exemplu:

LESS # ((x + 1) (2 * y)).

Scanned with OKEN Scanner

Predicatul LESS trebuie să compare rezultatul a două expresii, care mai întîi este necesar să fie evaluate: x + 1 și 2 * y

Formatul general este:

R # (E1 E2 ... En),

unde *R* reprezintă numele relației, iar E1, E2, ..., En sînt expresii. Semnul # arată că E1, E2, ..., En sînt expresii (aritmetice, apel de funcții) ce trebuie mai întîi evaluate.

Condiția de egalitate:

E1 = E2

este echivalentă cu expresia:

EQ # (E1 E2)

De exemplu,

LESS # ((x + 1) (2 * y))

se compilează ca:

(X LESS Y) # (+(x 1 X) and *(2 y Y))

și se citește astfel:

X mai mic decît *Y*, unde *X* este *x* + 1 și *Y*, 2 * *y*.

Semnul „#" se poate folosi ca o formă specială de întrebare pentru a evalua o expresie. Astfel, simbolul „#" urmat de o expresie furnizează ca răspuns valoarea expresiei. Exemplu:

```
&. # (2 * 4 + 5)
13
&.
```

Format general:

&. # (2 * 4 + 5)

Este o formă prescurtată a întrebării:

&.which(x : x = (2 * 4 + 5))

2.10. Utilizarea relației „is-told"

Is-told este o relație specială prin care utilizatorul răspunde la întrebări. Parametrul acestei relații specifică modelul de răspuns la întrebare.

Scanned with OKEN Scanner

Notă. Pentru utilizarea relației „is-told" este necesar ca programul SIMPLE să fie extins cu fișierul TOLD. Acesta se încarcă cu comanda „load TOLD".

Sintaxa relației *is-told* este următoarea:

<model-întrebare> is-told.

Elementele din modelul întrebării sînt afișate de către sistemul micro-PROLOG, fiind urmate de semnul întrebării (?), ce indică faptul că se așteaptă un răspuns. Răspunsurile posibile sînt următoarele:

- yes : relația *is-told* se consideră „adevărată";
- no : relația *is-told* se consideră „falsă";
- ans v1 v2 ..., unde v1, v2, ... reprezintă o secvență de valori, cîte una pentru fiecare variabilă din relația *is-told*. Aceste valori se atribuie variabilelor din condiția *is-told*, se evaluează expresia obținută și se afișează răspunsul. Apoi se repetă întrebarea;
- just v1 v2 ... : prelucrarea este aceeași ca mai sus, dar nu se mai repetă întrebarea.

Fie întrebarea:

&.all(X is Y la sută : (X) is-told & Y = (X/50*100))

```
X? ans 50
50 is 100 la sută
X? ans 25
25 is 50 la sută
X? just 10
10 is 20 la sută
No (more) answers
```

Se observă că *X* apare ca argument al relației *is-told*. El este afișat de sistemul micro-PROLOG, fiind urmat de semnul întrebării (?). Valorile introduse de utilizator sînt 50, 25, 10, iar răspunsurile furnizate de sistemul micro-PROLOG reprezintă calculul procentajului de 50%.

2.11. Comentarii

Comentariile se pot introduce în două moduri: prin utilizarea relației „/*" sau a faptului „comment".

Sintaxa relației „/*" este:

/* (comentariu)

Scanned with OKEN Scanner

Ea este igonorată de sistemul micro-PROLOG. Exemplu:

```
x factorial y if /*(x y trebuie să fie întregi)&
1 LESS x & y = (x*(factorial(x—1)))
```

În definiția factorialului se specifică prin relația /* că x și y trebuie să fie întregi. Viteza de execuție a unui program documentat este mai mică, deoarece sistemul micro-PROLOG, deși ignoră relațiile comentariu /*, trebuie totuși să le parcurgă.

Faptul „comment" permite specificarea de comentarii pentru un termen anume, dîndu-se posibilitatea să fie regăsite ulterior.

De exemplu, comentariul despre funcția factorial se poate specifica astfel:

factorial.comment (argumentele trebuie să fie întregi).

Regăsirea comentariului are loc după exemplul de mai jos:

```
&.which(x: factorial comment x)
(argumentele trebuie să fie întregi)
No (more) answers.
```

2.12. Prelucrarea listelor

Concatenarea a două liste: relația *append*, definită mai jos, concatenează două liste; *append* (*x y z*) înseamnă că lista *z* este formată din lista *x* urmată de lista *y*. Definiția predicatului *append* este:

```
append ( () x x)
append ((x|X) y (x|Z)) if append (X y Z)
```

Relația *append* se poate folosi de asemenea pentru a descompune o listă în sublistele sale. De exemplu,

```
&.which(x y: append (x y (1 2 3)))
() (1 2 3)
(1) (2 3)
(1 2) (3)
(1 2 3) ()
No (more) answers
```

Se pot selecta anumite subliste din mulțimea tuturor sublistelor. De pildă, se cere ca argumentul doi., din întrebarea de mai sus, să specifice numai sublistele ce încep cu 2:

```
&.which(x (2|Y) : append(x (2|Y) (1 2 3)))
(1) (2 3)
No (more) answers
```


Scanned with OKEN Scanner

Exemple de reguli care utilizează relația „append":

- *Cap de listă:*

```
cap(x y z) if append (y y1 Z) & y lungime x
```

Relația este adevărată dacă *y* conține primele *x* elemente din lista Z. Se poate utiliza pentru a găsi primele *x* elemente dintr-o listă. De exemplu,

```
&.which(x: cap(2 x (a b c d))
(a b)
No (more) answers.
```

- *Inversarea* unei liste:

```
(x) invers (x)
(x y|X) invers z if (y||X) invers Y & append (Y (x) z)
```

Lista *z* este inversă listei *x*. Exemplu de utilizare:

```
&.which(z: (1 2 3 4 5) invers z)
(5 4 3 2 1)
No (more) answers.
```

Relația „append" se poate declara și ca funcție:

```
function append
```

deoarece argumentul Z din „append(x y Z)" este determinat în mod unic de *x* și *y*. În acest caz *append* se poate apela ca o funcție.

Astfel, expresia:

```
&.#(append(a b) (append(c d) (e f)))
```

va produce rezultatul (a b c d e f)

Regula

```
(x|X) invers Z if X invers Y & Z=(append Y (x) )
```

apelează relația „append" ca o funcție.

2.13. Facilități generale

Obținerea unui singur răspuns

Pentru a obține un singur răspuns la o întrebare se folosește formatul prefix al relației, urmat de semnul exclamării (!):

```
R! (t1 t2 ... tn).
```


Scanned with OKEN Scanner

În momentul în care se obține o soluție, nu se mai continuă căutarea altor soluții. De exemplu, pentru a afla toți prietenii lui Dan se pune întrebarea:

&.which(x : ∪ x ∪ prieten dan).

Dar pentru a obține un singur prieten (primul din baza de date) trebuie pusă întrebarea:

&.which(x: prieten !(x,dan)).

O altă posibilitate pentru a împiedica operația de bactracking o constituie utilizarea caracterului slash (/) după relația ce trebuie satisfăcută o singură dată. În cazul de mai sus se pune următoarea întrebare:

&.which(x : ∪ x ∪ prieten dan & /).

Trebuie remarcat că primitiva / împiedică revenirea înapoi pentru orice relație care o precede.

Primitiva / se utilizează mai ales în reguli, și joacă același rol ca și operatorul „fail" (!).

Stiva întrebărilor

Pentru evaluarea unei întrebări, sistemul micro-PROLOG parcurge o mulțime de relații și fapte din baza de date, iar secvența utilizării acestora se ține prin construirea unei stive. De fiecare dată cînd o relație generează o nouă întrebare, se pune noua întrebare în vîrful stivei. Stiva pornește inițial cu o singură întrebare, cea pusă de utilizator. Pe măsura evaluării întrebării stiva crește. Stiva este utilizată și pentru revenirea înapoi *(bactracking)* în cazul unui insucces pe o anumită ramură. Dacă nu mai există spațiu de memorie pentru extinderea stivei, sistemul micro-PROLOG tipărește mesajul „no space left" și se forțează terminarea întrebării curente.

Înainte de relansarea programului trebuie eliminate definițiile inutile din baza de date a programului sau modulele de program ce nu sînt folosite. O altă soluție constă în scrierea recursivă la coadă a definițiilor astfel că spațiul utilizat de stivă se micșorează. De asemenea, primitiva / economisește spațiul stivei.

Definițiile recursive care economisesc spațiul stivei sînt acelea în care relația recursivă apare în ultima poziție a definiției regulii. (Aceste definiții se mai numesc și definiții recursive la coadă.)

Scanned with OKEN Scanner

Exemplu:

append (() x x)
append((x|y) z (x|y1)) if append (y z y1)

Formatul general al definiției recursive la coadă este:

R(t1 t2 ... tn) if C1 & C2 ...& Ck & R(s1 s2 ... sn).

Uneori definițiile recursive se pot transforma în definiții recursive la coadă, dacă se adaugă o relație mai generală. De exemplu, mai jos se prezintă un program pentru obținerea celui mai mare număr dintr-o listă nevidă, care conține o definiție recursivă, dar nu la coadă:

x max (x)
x max (y|Z) if x1 max Z & x mai-mare (y x1)
x mai-mare (x x)
x mai-mare (x y) if y less x
y mai-mare(x y) if x less y

Pentru a obține o relație recursivă, la coadă se definește o relație mai generală, x maximum (y Z), care exprimă faptul că *x* este mai mare ca *y* ,de asemenea fiind maximul listei Z:

y maximum (() y)
z maximum ((y1|Z y) if y2 mai-mare (y1 y)& / & z
maximum (Z y2)

Ultima regulă exprimă faptul că numărul mai mare ca *y* și maximul lui (y1|Z) este numărul mai mare decît maximul lui Z și *y* precum și y1.

Primitiva / după relația *mai-mare* indică sistemului micro-PROLOG că această relație are o singură soluție.

Se poate defini relația inițială „max" în funcție de „maximum":

x max (y|Z) if x maximum (Z y),

adică *x* este maximum lui (y|Z) dacă este mai mare ca *y* și maximum lui Z.

Utilizarea modulelor

Modulul este o colecție de definiții de relații cu două liste de nume: lista de export și, respectiv de import.

În lista de export se includ toate numele relațiilor din cadrul modulului, care se vor utiliza în cereri sau reguli din afara modulului.

În lista de import se introduc toate numele relațiilor ce sînt definite în afara modulului, dar sînt folosite în definiții din interiorul

Scanned with OKEN Scanner

modulului. Se mai includ de asemenea toate numele de termeni utilizați în interiorul modulului, dar care se folosesc în cereri cu relații de export. Odată construit un modul, nu se mai poate adăuga sau șterge nici una din relațiile de export ale modulului.

Toate numele care nu se află în lista de import sau export sînt locale (private) modulului. Dacă un nume local apare și în alt modul el va fi tratat de sitemul micro-PROLOG ca nume complet diferit.

Sistemul micro-PROLOG construiește un dicționar pentru numele locale și alt dicționar pentru numele din spațiul de lucru al programului. Aceasta înseamnă că definițiile relațiilor locale ale unui modul nu sînt accesibile altui modul.

Așa cum se știe, definițiile introduse prin comenzile „accept" și „add" sau care se introduc în memorie o dată cu încărcarea programului se păstrează într-o zonă specială, numită „spațiu de lucru". Comanda „list" afișează numai definițiile din această zonă. Modulele ce se încarcă ulterior nu intră în spațiul de lucru, deși ocupă o zonă din memorie.

Fie de exemplu programul următor, care sortează o listă de numere întregi în ordine crescătoare:

```
() sort ()
(x) sort (x)
(x1 x2|x) sort y if
            împarte ((x1 x2|x x1 x2) &
            x1 sort y1 & x2 sort y2 & / &
            unește (y1 y2 y)
unește (()x x)
unește (x () x)
unește (x1|y1) (x2|y2) (x|y)) if
            alege! ((x1|y1) (x2|y2) x y1 y2)&
            unește (y1 y2 y)
alege ((x1|y1)  (x2|y2) x1 y1 (x2|y2)) if
            x1 less x2
alege ((x1|y1) (x2|y2) x2(x1|y1) y2) if
            not x1 less x2
împarte (X X1 X2) if
            diviz # ((div (lungimex) 2) X X1 X2)
diviz (0 X () X)
diviz (y (x|X) (x|X1) X2) if
            O LESS y &
            SUM (y1 1 y) &
            diviz (y X X1 X2)
```


Scanned with OKEN Scanner

Relația „sort" este principală, celelalte fiind necesare numai procesului de sortare. Dacă se construiește un modul din acest program este suficient să se includă în lista de export doar relația „sort". Relația „less" trebuie inclusă în lista de import, deoarece ea nu este definită în interiorul modulului. De asemenea, relațiile „div" și „lungime".Relațiile SUM și LESS sînt primitive ale limbajului micro-PROLOG, deci nu este necesară includerea lor în lista de import. În schimb, numele # trebuie importat dintr-un modul al preprocesorului SIMPLE.

Referitor la construirea unui modul, acest aspect se va exemplifica în programul „sort". În primul rînd trebuie încărcate în spațiul de lucru (introduse prin comanda „add" sau încărcate de pe fișiere) toate definițiile utilizate în modul. În cazul modulului sort avem definițiile:

```
() lungime 0
(x|y) lungime z if y lungime z1 & SUM (1 z1 z)
div(x y z) if TIMES (y z1 x) & INT(z1 z)
```

În continuare, se încarcă un utilitar, numit MODULES, el însuși un modul, furnizat o dată cu sistemul micro-PROLOG:

```
&.load MODULES
```

Acest program permite construirea și salvarea unui modul utilizator. Apoi se introduce comanda de definire a modulului:

```
&.Module(sort-mod (sort) (less #))
```

Relația are trei argumente: numele modulului (sort-mod); lista de export (sort); lista de import (less, #).

Se introduce apoi comanda:

```
&.wrap SORT
```

În felul acesta programul „sort" va fi construit ca un modul și salvat în fișierul SORT, într-un format special. Spațiul de lucru se șterge. Numele fișierului trebuie să fie diferit de numele dat modulului și fiecare din acestea să difere de numele relațiilor din modul.

Comanda „wrap" utilizează informații furnizate de comanda „module" (de exemplu, pentru a construi listele numelor de import și export).

Dacă se dorește folosirea programului „sort" se utilizează comanda:

```
&.load SORT
```

Deoarece programul este un modul el nu va intra în întregime în spațiul de lucru; ca urmare, o comandă „list all" nu-l va lista.

Scanned with OKEN Scanner

Pentru a verifica dacă a fost încărcat se utilizează comanda:

```
&.is (sort defined)
```

Dacă există deja o relație cu numele „sort" în spațiul de lucru al programului, atunci la comanda load SORT se obține un mesaj de eroare („utilizare ilegală de module)" și operația de încărcare nu se mai realizează. De asemenea, dacă se încearcă introducerea unei relații cu numele „sort" după încărcarea modulului, se obține un mesaj de eroare: „nu se pot adăuga relații pentru sort", motivul constînd în aceea că numele relațiilor exportate sînt protejate.

Înainte de folosirea relației „sort" se impune introducerea în spațiul de lucru a definițiilor pentru numele importate (în cazul studiat „less": x less y if x LESS y).

În continuare, se poate utiliza relația „sort" pentru sortarea unei liste, de exemplu:

```
&.which(x: (4 3 5 6 2) sort x)
(2 3 4 5 6 )
No (more) answers
```

Dacă nu se mai utilizează relația „sort", atunci se poate elimina modulul „sort-mod" din spațiul de lucru și din memorie:

```
&.kill sort-mod
```

Citirea de la terminal

Introducerea de termeni de la terminal se realizează cu primitiva *R*. Formatul ei este R(x), unde *x* este o variabilă neinițializată. În momentul în care primitiva se evaluează, se afișează la terminal prompterul „." și se așteaptă introducerea termenului. Variabila *x* primește ca valoare termenul respectiv. Termenul poate fi un număr, o constantă, o listă sau o variabilă. Sfîrșitul introducerii termenului se specifică prin apăsarea tastei ENTER.

La o cerere a primitivei *R* se pot introduce mai mulți termeni, separați prin virgulă (ultimul termen este urmat de ENTER). În acest caz doar primul termen este utilizat pentru a satisface primitiva curentă *R*. Următoarea primitivă *R* nu va mai cere introducerea termenului de la terminal, ci va utiliza al doilea termen din lista precedentă, și așa mai departe pînă la epuizarea listei.

Constantele, ce conțin caractere speciale, se includ între ghilimele. De exemplu: „continuă prelucrarea" este o constantă care conține caracterul special spațiu; ea se poate introduce la o cerere a unei primitive *R*.

Scanned with OKEN Scanner

Scrierea la terminal

Scrierea unui termen la display se realizează cu primitiva *P*. Formatul este P(t1 t2 ... tn), unde t1, t2, ..., tn sînt termeni ce se afișează la terminal .

O altă primitivă de scriere la terminal, echivalentă cu *P*, este primitiva *PP*. Spre deosebire de *P* ea readuce cursorul la început de linie, după ce a afișat termenii, și include între ghilimele toate constantele ce au fost introduse de la terminal între ghilimele.

Exemple:

- P(„continuă prelucrarea") afișează continuă prelucrarea;
- PP(„continuă prelucrarea") afișează „continuă prelucrarea".

Exemplu de utilizare a primitivelor *R* și *P*:

x listă if P(introduceți o listă) & R(y) & P (y)

La întrebarea:

&.which(x : x listă)

se obține următorul răspuns:

introduceți o listă. (2 5 7 9)
(2 5 7 9)
No (more) answers.

Variabilele furnizate ca termeni la o cerere a primitivei *R* sînt convertite într-un format intern, astfel că numele lor se pierde. Dacă se dorește listarea unor variabile cu primitiva *P* sau *PP* se vor atribui ca nume externe noi nume, astfel că prima variabilă se va numi *X*, a doua *Y* ș.a.m.d., ca în secvența de mai jos:

X Y Z x y z X1 Y1 .....

La fiecare apel al lui *P* sau *PP* afectarea numelor variabilelor se face de la începutul listei de mai sus. Astfel, poate exista situația încare două variabile diferite au același nume (afișat la consolă). De exemplu.

&.is (PP(x) & P(y))
X
X
YES

Primitivele *R*, *P*, *PP* stau la baza realizării comenzilor predefinite *accept*, *add*, *save*, *load* etc. Ele se pot utiliza de asemenea pentru definirea altor comenzi.

Scanned with OKEN Scanner

2.14. Concluzie

Micro-PROLOG — o variantă a limbajului PROLOG, implementată pe microcalculatoare — asigură definirea de reguli și fapte, construirea de baze de date, prelucrarea listelor și alte prelucrări complexe utilizate în inteligența artificială și, în particular, în domeniul sistemelor-expert. Spre deosebire de limbajul PROLOG standard, varianta micro-PROLOG prezentată are mai multe caracteristici imperative: comenzi pentru adăugarea sau ștergerea de instrucțiuni din program, editarea instrucțiunilor, citirea sau scrierea de la terminal sau dintr-un fișier etc. Pe de altă parte în programul micro-PROLOG se pot defini noi comenzi, astfel încît utilizatorul își poate construi aplicația în conformitate cu cerințele sale și mediul respectiv corespunzător dezvoltării programelor.

Prin micro-PROLOG, microcalculatoarele devin disponibile pentru elaborarea de aplicații corespunzătoare noului stil de programare — programarea logică.

Scanned with OKEN Scanner

ANEXA III.1.

PREDICATE PREDEFINITE ÎN PROLOG

Sînt descrise pe scurt predicatele predefinite ce se întîlnesc în majoritatea implementărilor limbajului PROLOG. Aceste predicate se referă la operații de intrare-ieșire, operații aritmetice și relaționale, controlul execuției programului, manipularea bazei de date a programului. Se întîlnesc de asemenea predicate specifice mediului de programare, în PROLOG relative la manipularea programului PROLOG (salvare, restaurare, conversie caractere etc):

abort:	termină execuția programului PROLOG;
arg(I,T,X)	*I* este un număr întreg pozitiv, iar *T* — un termen compus. *X* primește valoarea argumentului numărul *I* al termenului *T* (numerotarea argumentelor începe de la 1);
assert(C)	*C* este o clauză ce se adaugă programului PROLOG într-o poziție dependentă de implementare. *C* nu poate fi variabilă;
asserta(C)	ca și *assert*(*C*), dar clauza se adaugă la începutul programului;
assertz(C)	ca și *assert* (*C*), dar clauza se adaugă la sfîrșitul programului;
atom(X)	testează dacă *X* este inițializat cu un atom;
atomic(X)	testează dacă *X* este inițializat cu un atom sau un întreg;
break:	întrerupe execuția programului la următorul apel de procedură și apare la terminal mesajul "% break:". Interpretorul poate accepta comenzi specifice mediului de programare PROLOG (statistici, validare trasare etc). Reluarea execuției se face prin introducerea comenzii „:—end." sau „^Z" Se poate termina execuția programului cu comandă „:-abort.";
call(X)	*X* trebuie să fie inițilizat cu un termen ce reprezintă corpul unei clauze. Această clauză se execută ca și cum ar apărea ca text în locul apelului *call* (*X*);
clause(P,Q)	se caută în baza de date a programului clauza de nume (cap) *P*. Se atribuie lui *Q* corpul clauzei găsite. Dacă *P* nu are corp, atunci *Q* primește valoarea „true";
close(F)	se închide fișierul specificat de *F*;
consult(F)	se cere interpretorului să citească din fișierul F. Dacă se întîlnește o directivă pentru interpretor (de exemplu, 'NOLC') aceasta se execută imediat. Dacă se citește o clauză, aceasta se adaugă bazei de date a programului (în secvența citirii). Prin acest predicat se inserează text (proceduri) în programul PROLOG din alte fișiere;
depth(D)	*D* primește ca valoare nivelul curent de imbricare, ca rezultat al apelurilor de proceduri (este număr întreg);

Scanned with OKEN Scanner

display(X)	se scrie la terminal termenul X în notație prefixată parantezată;
erase(P)	se șterge din baza de date termenul a cărui adresă se află în P (vezi *recorded* (X, P));
eraseall(X)	se șterg din baza de date toți termenii identici cu X;
fail	furnizează întotdeauna insucces;
fileerrors	suprimă efectul predicatul „nofileerrors", adică în caz de eroare de intrare-ieșire se afișează mesajul la terminal și se apelează „abort";
functor(T,F,N)	T este o structură sau un atom, iar F și N sînt variabile. Dacă T este atom, atunci F ia numele atomului și N devine 0. Dacă T este o structură, atunci F ia numele structurii, iar N conține numărul de argumente;
gc	validează „garbage collection" în stiva globală;
get(N)	se citește caracterul următor non-blank și tipăribil din fișierul curent de intrare și se atribuie lui N codul ASCII al caracterului citit;
geto(N)	se citește caracterul următor din fișierul curent de intrare și se atribuie lui N codul ASCII al caracterului;
gcguide(N)	N este un număr întreg de la 0 la 512 și indică limita de pagini de stivă globală sub care nu se mai face „garbage collection";
instance(P,X)	lui X i se atribuie termenul a cărui adresă se află în P (vezi *recorded* (P, X));
integer(X)	testează dacă X este inițializat cu un întreg;
'LC'	validează recunoașterea întregului set de caractere (se consideră prin lipsă);
leash	validează modul „leashed" pentru trasare (se consideră prin lippsă);
length(L,N)	L trebuie să desemneze o listă. Lui N i se atribuie lungimea listei;
listing	se listează toate clauzele programului în fișierul curent de ieșire;
listing(A)	listează în fișierul curent de ieșire toate clauzele cu nume A;
log	validează înregistrarea acțiunilor de la terminal în fișierul PROLOG.LOG (se consideră prin lipsă);
maxdepth(D)	D este un întreg pozitiv ce limitează numărul maxim de imbricări generate de apelurile de proceduri. Nivelul cel mai de sus corespunde valorii zero. Acest predicat este util pentru a evita umplerea stivei prin apeluri de proceduri într-o buclă infinită;
name(X,L)	Dacă X este un atom sau un întreg, atunci L este o listă formată din codurile ASCII ale caracterelor numelui lui X;
nofilleerrors	invalidează listarea la terminal a mesajelor de eroare pentru operațiile de intrare-ieșire. În caz de eroare se face apel la „fail". Prin lipsă se consideră „fileerrors", adică se afișează mesajul de intrare-ieșire și se face apel la „abort";
nl	se face salt la o nouă linie în fișierul curent de ieșire;
nogc	invalidează acțiunea de „garbage collection" în stiva globală;
'NOLC'	stabilește convenția „fără litere mici";
nolog	invalidează înregistrarea acțiunilor de la terminal în fișierul PROLOG.LOG;
nonvar(X)	testează dacă X este inițializat cu un termen ce nu este variabilă;
not(P)	dacă P are soluție se furnizează insucces („fail"), altfel — succes. Se definește astfel:

```
not(P) :- P,!,fail.
not(_).
```

notrace	invalidează trasarea (se consideră prin lipsă);
numbervars(X,N,M)	atribuie fiecărei variabile din termenul X un termen special, astfel încît „write (X)" tipărește fiecare din aceste variabile ca „I"

Scanned with OKEN Scanner

unde „I-"urile sînt întregi consecutivi de la N la M—1. N și *M*, trebuie să fie inițializați cu întregi;

put(N) — se scrie în fișierul curent de ieșire caracterul ASCII cu codul *N*. *N* poate fi și o expresie de tip întreg;

putatom(X) — se scrie în fișierul curent de ieșire numele atomului *X*;

read(X) — *X* este un termen căruia i se atribuie entitatea citită din fișierul curent de intrare (delimitată de un separator, ce poate fi punctul „." urmat de „carriage return" sau „spațiu"). Dacă se detectează sfîrșitul de fișier, *X* primește termenul „:-end.". După ce s-a detectat sfîrșitul de fișier următoarele apeluri la *read* se termină cu eroare;

reconsult(F) — *F* este un specificator de fișier. Are același efect ca și predicatul *consult* (*F*), cu deosebirea că acele clauze care există deja în baza de date se înlocuiesc cu cele din fișierul *F*, Se folosește pentru actualizarea sau depanarea programelor deja existente. Dacă se specifică *reconsult* (*user*) se vor introduce modificările direct de la terminal; sfîrșitul introducerii se specifică cu „:-end." sau ^Z;

record(X) — *X* se scrie în baza de date (într-o poziție dependentă de implementarea sistemului PROLOG). *X* nu poate fi variabilă;

recorda(X) — ca și *record* (*X*), dar termenul *X* se scrie la începutul bazei de date;

recorded(X,P) — se caută în baza de date termenul anterior cu nume identic cu *X*. Termenul găsit se actualizează cu noua valoare a lui *X* și se rescrie în baza de date. *P* este un *pointer* ce conține adresa termenului actualizat;

recordz(X) — ca și *record* (*X*), dar termenul *X* se scrie la sfîrșitul bazei de date;

rename(F,N) — se redefinește fișierul *F* cu numele *N*. Dacă fișierul era deschis, mai întîi se închide și apoi se redefinește. Dacă *N* este '[ ]' se șterge fișierul *F*.

repeat — generează un ciclu infinit;

restore(F) — restaurează sistemul salvat în fișierul *F* cu comanda *save*;

retract(C) — se șterge clauză identică ca nume cu *C*. *C* ia valoarea clauzei șterse (*C* nu poate fi variabilă). Căutarea clauzei se face de la începutul programului (la o reluare prin *backtraking* se șterge următoarea clauză cu nume identic cu *C*);

retractall(C) — se șterg toate clauzele din program al căror nume este identic cu *C* (*C* nu poate fi variabilă).

save(F) — se salvează starea curentă a sistemului în fișierul *F*;

see(F) — fișierul *F* devine fișier curent de intrare;

seen: — închide fișierul curent de intrare;

seeing(F) — *F* primește numele fișierului curent de intrare;

skip(N) — se sare în fișierul curent de intrare după următorul caracter cu codul ASCII *N*. *N* poate fi o expresie de tip întreg;

statistics — afișează la terminal date statistice privind utilizarea memoriei, timpul de execuție, „garbage collection" în stiva globală etc.

statistics(K,V) — permite programului să obțină diverse date statistice în funcție de cuvîntul cheie *K*. Valorile statistice se atribuie lui *V*. Mai jos se prezintă cuvintele cheie pe care le poate lua *K* și valorile statistice corespunzătoare atribuite lui *V*.

Timpul se reprezintă în milisecunde, iar spațiul de memorie în cuvinte. Dacă valoarea timpului depășește 129071 secunde se va reprezenta astfel:

%wd(T1, T2)

și semnifică valoarea: T1*2^18 + T2 mod 2^18.

Scanned with OKEN Scanner

display(X)	se scrie la terminal termenul X în notație prefixată parantezată;
erase(P)	se șterge din baza de date termenul a cărui adresă se află în P (vezi *recorded* (X, P));
eraseall(X)	se șterg din baza de date toți termenii identici cu X;
fail	furnizează întotdeauna insucces;
fileerrors	suprimă efectul predicatul „nofileerrors", adică în caz de eroare de intrare-ieșire se afișează mesajul la terminal și se apelează „abort";
functor(T,F,N)	T este o structură sau un atom, iar F și N sînt variabile. Dacă T este atom, atunci F ia numele atomului și N devine 0. Dacă T este o structură, atunci F ia numele structurii, iar N conține numărul de argumente;
gc	validează „garbage collection" în stiva globală;
get(N)	se citește caracterul următor non-blank și tipăribil din fișierul curent de intrare și se atribuie lui N codul ASCII al caracterului citit;
geto(N)	se citește caracterul următor din fișierul curent de intrare și se atribuie lui N codul ASCII al caracterului;
gcguide(N)	N este un număr întreg de la 0 la 512 și indică limita de pagini de stivă globală sub care nu se mai face „garbage collection";
instance(P,X)	lui X i se atribuie termenul a cărui adresă se află în P (vezi *recorded* (P, X));
integer(X)	testează dacă X este inițializat cu un întreg;
'LC'	validează recunoașterea întregului set de caractere (se consideră prin lipsă);
leash	validează modul „leashed" pentru trasare (se consideră prin lippsă);
length(L,N)	L trebuie să desemneze o listă. Lui N i se atribuie lungimea listei;
listing	se listează toate clauzele programului în fișierul curent de ieșire;
listing(A)	listează în fișierul curent de ieșire toate clauzele cu nume A;
log	validează înregistrarea acțiunilor de la terminal în fișierul PROLOG. LOG (se consideră prin lipsă);
maxdepth(D)	D este un întreg pozitiv ce limitează numărul maxim de imbricări generate de apelurile de proceduri. Nivelul cel mai de sus corespunde valorii zero. Acest predicat este util pentru a evita umplerea stivei prin apeluri de proceduri într-o buclă infinită;
name(X,L)	Dacă X este un atom sau un întreg, atunci L este o listă formată din codurile ASCII ale caracterelor numelui lui X;
nofilleerrors	invalidează listarea la terminal a mesajelor de eroare pentru operațiile de intrare-ieșire. În caz de eroare se face apel la „fail". Prin lipsă se consideră „fileerrors", adică se afișează mesajul de intrare-ieșire și se face apel la „abort";
nl	se face salt la o nouă linie în fișierul curent de ieșire;
nogc	invalidează acțiunea de „garbage collection" în stiva globală;
'NOLC'	stabilește convenția „fără litere mici";
nolog	invalidează înregistrarea acțiunilor de la terminal în fișierul PROLOG.LOG;
nonvar(X)	testează dacă X este inițializat cu un termen ce nu este variabilă;
not(P)	dacă P are soluție se furnizează insucces („fail"), altfel — succes. Se definește astfel: `not(P) :- P,!,fail.` `not(_).`
notrace	invalidează trasarea (se consideră prin lipsă);
numbervars(X,N,M)	atribuie fiecărei variabile din termenul X un termen special, astfel încît „write (X)" tipărește fiecare din aceste variabile ca „I"

Scanned with OKEN Scanner

unde „I-"urile sînt întregi consecutivi de la N la M−1. N și M, trebuie să fie inițializați cu întregi;

put(N) se scrie în fișierul curent de ieșire caracterul ASCII cu codul N. N poate fi și o expresie de tip întreg;

putatom(X) se scrie în fișierul curent de ieșire numele atomului X;

read(X) X este un termen căruia i se atribuie entitatea citită din fișierul curent de intrare (delimitată de un separator, ce poate fi punctul „." urmat de „carriage return" sau „spațiu"). Dacă se detectează sfîrșitul de fișier, X primește termenul „:-end.". După ce s-a detectat sfîrșitul de fișier următoarele apeluri la *read* se termină cu eroare;

reconsult(F) F este un specificator de fișier. Are același efect ca și predicatul *consult* (F), cu deosebirea că acele clauze care există deja în baza de date se înlocuiesc cu cele din fișierul F, Se folosește pentru actualizarea sau depanarea programelor deja existente. Dacă se specifică *reconsult* (*user*) se vor introduce modificările direct de la terminal; sfîrșitul introducerii se specifică cu „:-end." sau ^Z;

record(X) X se scrie în baza de date (într-o poziție dependentă de implementarea sistemului PROLOG). X nu poate fi variabilă;

recorda(X) ca și *record* (X), dar termenul X se scrie la începutul bazei de date;

recorded(X,P) se caută în baza de date termenul anterior cu nume identic cu X. Termenul găsit se actualizează cu noua valoare a lui X și se rescrie în baza de date. P este un *pointer* ce conține adresa termenului actualizat;

recordz(X) ca și *record* (X), dar termenul X se scrie la sfîrșitul bazei de date;

rename(F,N) se redefinește fișierul F cu numele N. Dacă fișierul era deschis, mai întîi se închide și apoi se redefinește. Dacă N este '[ ]' se șterge fișierul F.

repeat generează un ciclu infinit;

restore(F) restaurează sistemul salvat în fișierul F cu comanda *save*;

retract(C) se șterge clauză identică ca nume cu C. C ia valoarea clauzei șterse (C nu poate fi variabilă). Căutarea clauzei se face de la începutul programului (la o reluare prin *backtraking* se șterge următoarea clauză cu nume identic cu C);

retractall(C) se șterg toate clauzele din program al căror nume este identic cu C (C nu poate fi variabilă).

save(F) se salvează starea curentă a sistemului în fișierul F;

see(F) fișierul F devine fișier curent de intrare;

seen: închide fișierul curent de intrare;

seeing(F) F primește numele fișierului curent de intrare;

skip(N) se sare în fișierul curent de intrare după următorul caracter cu codul ASCII N. N poate fi o expresie de tip întreg;

statistics afișează la terminal date statistice privind utilizarea memoriei, timpul de execuție, „garbage collection" în stiva globală etc.

statistics(K,V) permite programului să obțină diverse date statistice în funcție de cuvîntul cheie K. Valorile statistice se atribuie lui V. Mai jos se prezintă cuvintele cheie pe care le poate lua K și valorile statistice corespunzătoare atribuite lui V.

Timpul se reprezintă în milisecunde, iar spațiul de memorie în cuvinte. Dacă valoarea timpului depășește 129071 secunde se va reprezenta astfel:

%wd(T1, T2)

și semnifică valoarea: T1*2^18 + T2 mod 2^18.

Scanned with OKEN Scanner

K	V		
core	segment inferior	segment superior	—
global-stack	dimensiune	spațiu liber	—
local-stack	dimensiune	spațiu liber	—
runtime	timpul de lansare sistem PROLOG	timpul de la ultima statistică	—
garbage-collection	număr de garbage-collection	spațiu eliberat	timp consumat

tab(N) — se tipăresc N spații în fișierul curent de ieșire. N poate fi o expresie de tip întreg;

tell(F) — fișierul F devine fișier curent de ieșire;

telling(F) — F primește numele fișierului curent de ieșire;

told — închide fișierul curent de ieșire; told

trace — validează trasarea (pentru depanare).

ttyflush — se șterge zona tampon de ieșire a terminalului;

ttynl — se face salt la o nouă linie la terminal și se șterge zona tampon de ieșire;

ttyput(N) — se scrie la terminal caracterul cu codul ASCII N N poate fi o expresie de tip întreg;

ttyset(N) — N este codul ASCII al următorului caracter tipăribil și diferit de spațiu introdus de la terminal;

ttyseto(N) — N este codul ASCII al următorului caracter introdus de la terminal;

ttyskip(N) — se sare după ultimul caracter cu codul ASCII N introdus de la terminal. N poate fi expresie de tip întreg;

var(X) — testează dacă X este inițializat cu o variabilă;

write(X) — se scrie termenul X în fișierul curent de ieșire.

X — are același efect ca și *call* (X);

! — tăietura;

X + Y — adunare;

X − Y — acădere;

X * Y — înmulțire;

X/Y — împărțire;

X mod Y — X modulo Y;

−X — minus;

X /\ Y — conjuncție (și) pe bit;

X \/ Y — disjuncție (sau) pe bit;

\X — complement;

Scanned with OKEN Scanner

X < < Y	deplasarea lui *X* spre stînga cu *Y* poziții;
X > > Y	deplasarea lui *X* spre dreapta cu *Y* poziții;
!(X)	număr între 0 și 2^18 — 1 egal cu X modulo 2 18;
$(X)	număr între —2^17 și 2^17 — 1 egal cu X modulo 2^18;
Z is X	*X* este o expresie de tip întreg, care se evaluează și se atribuie lui *Z*;
X< Y	mai mic;
X> Y	mai mare;
X =< Y	mai mic sau egal;
X >= Y	mai mare sau egal;
X == Y	*X* este identic cu *Y*;
X \== Y	*X* nu este identic cu *Y*;
X =.. Y	univ;
X,Y	*X* și *Y*;
X;Y	*X* sau *Y*.

Scanned with OKEN Scanner

BIBLIOGRAFIE

1. CLOCKSIN, W.F.; MELLISH, C.S., *Programming in PROLOG*, Springer-Verlag, Berlin, Heidelberg, New-York, 1981.

2. CLOCKSIN, W.; MELLISH, C., *The UNIX PROLOG System, Software Report 5*, University of Edinburg, Scotland, 1979.

3. PEREIRA, L.M.; PEREIRA, F.; WARREN, D., *User's guide to DECsystem PROLOG*, Laboratorio Nacional de Engenharia Civil, Lisabona, 1979.

4. CLOCKSIN, W.; MELLISH, C.; FISHER, R., *The RT-11 PROLOG System, Software Report 5a*, University of Endinburgh, Scotland, 1980.

5. KOWALSKI, R., *Logic for Problem Solving*, Amsterdam, North Holland, 1979.

6. CLARK, K.L.; McGABE, G.; ENNALS, J.R., *„ZX Spectrum — Micro-PROLOG primer*, Sinclair Research Ltd, England, 1983.

7. SANTANE-TOTH, E.; SZEREDI, P., *PROLOG applications in Hungary*, Logic Programming, Academic Press, New York.

8. MICHIE, D., *Expert System*, Computer Journal, 23, 1980.

9. WARREN, D.H.D.; PEREIRA, F.C.N., *An efficient easily adaptable system for interpreting natural language queries, Research Paper 155*, University of Edinburgh, Scotland, 1981.

10. PEREIRA, L.M.; SABATIER, P.; d'OLIVEIRA, E., *ORBI — An expert system for environmental resource evaluation through natural language*, Report FCT/DI-3/82, Universidade Nova de Lisboa, Portugal, 1982.

11. POPA, V.; STOICA, N.; PALADE, I.C., *MIX-PROLOG*, manual de utilizare, ICSIT-TCI București, 1988.

Scanned with OKEN Scanner

CUPRINS

Prefață 5
PARTEA I. PROGRAMARE ÎN LIMBAJ C 9

Capitolul 1. Introducere 9
Capitolul 2. Introducere în programarea în C 12

2.1. Structura sintetică a unui program C 12
2.2. Exemple de programe C 13
2.3. Reorientarea intrărilor și ieșirilor standard într-un program C 15
2.4. Variabile, expresii, și atribuiri 17
2.5. Constante simbolice 19

Capitolul 3. **Elemente de baza ale programării în C** 20

3.1. Operații de ieșire 21
3.2. Tipuri de date 23
3.3. Identificatori și cuvinte-cheie 28
3.4. Constante 29
3.5. Declarații 32
3.6 Operatori și expresii de asignare 32
3.7 Operatori aritmetici 33
3.8 Operatori de incrementare și decrementare 34
3.9 Operatori pentru prelucrare biți 35
3.10 Operatori de tip adresă 36
3.11 Operații de intrare 37
3.12 Operatori relaționali 40
3.13. Operatori și expresii logice 41
3.14 Precedența și asociativitatea operatorilor 42
3.15 Instrucțiuni și blocuri 43

Scanned with OKEN Scanner

3.16 Instrucțiunea *if* 43
3.17 Instrucțiunea *while* 46
3.18 Instrucțiunea *for* 48
3.19 Operatorul „virgulă" 50
3.20 Instrucțiunea *do-while* 51

Capitolul 4. **Programare avansată în C** 52

4.1 Funcții C 52
4.2 Variabile externe și variabile interne 56
4.3 Funcții recursive 59
4.4 Expresii condiționale 60
4.5 Selecție multiplă folosind instrucțiunea *switch* 61
4.6 Instrucțiunile *break* și *continue* 64
4.7 Pointeri 64
4.8 Vectori de pointeri 66
4.9 Funcții care returnează pointeri 67

Bibliografie 68

PARTEA a II-a. PROGRAMARE ÎN MODULA-2 69

Capitolul 1. **Lexicul limbajului Modula-2** 69

1.1 Noțiuni de modularizare cu MODULA-2 69
1.2 Exemplu de modularizare a unui program 73
1.3 Elementele unui program MODULA-2 75

Capitolul 2. **Variabile, expresii, instrucțiuni** 7

2.1 Declarații 77
2.2 Constante 78
2.3 Variabile 79
2.4 Expresii 79
2.5 Asignări 80
2.6 Structuri de control 81

Capitolul 3. **Tipuri de date predefinite** 86

3.1 Clasificarea tipurilor 87
3.2 Declararea tipurilor 87
3.3 Tipuri elementare 88
3.4 Tipul BITSET 93

Scanned with OKEN Scanner

Capitolul 4. Tipuri de date structurate 94

4.1 Tipuri de date enumerative 94
4.2 Tipuri subdomeniu 95
4.3 Tipuri SET 96
4.4 Tipuri de date ARRAY 99
4.5 Tipuri de date RECORD 101

Capitolul 5. Proceduri 108

5.1 Declararea procedurilor 108
5.2 Apelul unei proceduri 111
5.3 Instrucțiunea RETURN 113
5.4 Vizibilitatea identificatorilor 116
5.5 Recursivitatea procedurilor 117

Capitolul 6. Module 118

6.1 Module locale-vizibilitate 118
6.2 Corpuri de module locale 121
6.3 Module globale 128
6.4 Reguli de compilare a modulelor 135

Capitolul 7. Programarea sistemelor 136

7.1 Tipuri de date de acces 136
7.2 Operații cu date ale unui tip de acces 138
7.3 Alocarea și dezalocarea resurselor 140
7.4 Tipuri procedurale 143
7.5 Tipuri de date definite în modulul global SYSTEM 145
7.6 Procese concurente 148
7.7 Corutine 150
7.8 Localizarea variabilelor 152
7.9 Condiții de relaxare a tipizării 153
7.10 Drivere pentru periferice 155

ANEXA II.1. Sintaxa limbajului MODULA-2 159

ANEXA II.2. Secțiunea de definire a unor module globale puse la dispoziție de limbajul MODULA-2 161

Bibliografie 171

Scanned with OKEN Scanner

PARTEA a III-a PROGRAMARE ÎN PROLOG 172

Capitolul 1. Introducere în PROLOG 172

1.1 Introducere 172
1.2 Noțiunile de bază ale limbajului PROLOG 174
1.3 Elementele limbajului PROLOG 179
1.4 Operatori 180
1.5 Lista 182
1.6 Căutarea în baza de date 185
1.7 Tăietura 187
1.8 Intrări-ieșiri 188
1.9 Predicate predefinite 195
1.10 Aplicații 204
1.11 Depanarea programelor PROLOG 212
1.12 Versiuni ale limbajului PROLOG 216
1.13 Concluzii 218

Capitolul 2. Micro-PROLOG 219

2.1 Observații generale 219
2.2 Comenzi pentru modificarea bazei de date 222
2.3 Întrebări 223
2.4 Relații aritmetice 225
2.5 Reguli 228
2.6 Lista 228
2.7 Predicate predefinite 231
2.8 Expresii de apel de funcții 234
2.9 Condiția de egalitate 235
2.10 Utilizarea relației „is-told" 236
2.11 Comentarii 237
2.12 Prelucrarea listelor 238
2.13 Facilități generale 239
2.14 Concluzie 246

Anexa III.1. Predicate predefinite în PROLOG 247

Bibliografie 252

Redactor : maior dr. ing. ALEXANDRU MIHALCEA
Tehnoredactor : DIONISIE BODEA

Bun de tipar : 17.12.1989. Apărut : 1989.
Coli tipar : 16. B 10389.

Tiparul executat sub comanda
nr. 347
Întreprinderea poligrafică
„13 Decembrie 1918"
str. Grigore Alexandrescu nr. 89-97
București — România